

Extended Probabilistic Symbolic Execution

by

Aline Uwimbabazi

*Thesis presented in partial fulfilment of the requirements for
the degree of Master of Science at Stellenbosch University*



Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Professor Willem Visser

December 2013

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2013

Copyright © 2013 Stellenbosch University
All rights reserved.

Abstract

Extended Probabilistic Symbolic Execution

Aline Uwimbabazi

*Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.*

Thesis: MSc

December 2013

Probabilistic symbolic execution is a new approach that extends the normal symbolic execution with probability calculations. This approach combines symbolic execution and model counting to estimate the number of input values that would satisfy a given path condition, and thus is able to calculate the execution probability of a path. The focus has been on programs that manipulate primitive types such as linear integer arithmetic in object-oriented programming languages such as Java. In this thesis, we extend probabilistic symbolic execution to handle data structures, thus allowing support for reference types. Two techniques are proposed to calculate the probability of an execution when the programs have structures as inputs: an approximate approach that assumes probabilities for certain choices stay fixed during the execution and an accurate technique based on counting valid structures. We evaluate these approaches on an example of a Binary Search Tree and compare it to the classic approach which only take symbolic values as input.

Uittreksel

Uitgebreide Probabilistiese Simboliese Uitvoering

(“ *Extended Probabilistic Symbolic Execution* ”)

Aline Uwimbabazi

*Departement Wiskundige Wetenskappe,
Afdeling Rekenaarwetenskap,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.*

Tesis: MSc

Desember 2013

Probabilistiese simboliese uitvoering is ’n nuwe benadering wat die normale simboliese uitvoering uitbrei deur waarsynlikheidsberekeninge by te voeg. Hierdie benadering kombineer simboliese uitvoering en modeltellings om die aantal invoerwaardes wat ’n gegewe padvoorwaarde sal bevredig, te beraam en is dus in staat om die uitvoeringswaarsynlikheid van ’n pad te bereken. Tot dus vër was die fokus op programme wat primitiewe datatipes manipuleer, byvoorbeeld lineêre heelgetalrekenkunde in objek-geörienteerde tale soos Java. In hierdie tesis brei ons probabilistiese simboliese uitvoering uit om datastrukture, en dus verwysingstipes, te dek. Twee tegnieke word voorgestel om die uitvoeringswaarsynlikheid van ’n program met datastrukture as invoer te bereken. Eerstens is daar die benaderingstegniek wat aanneem dat waarsynlikhede vir sekere keuses onveranderd sal bly tydens die uitvoering van die program. Tweedens is daar die akkurate tegniek wat gebaseer is op die telling van geldige datastrukture. Ons evalueer hierdie benaderings op ’n voorbeeld van ’n binêre soekboom en vergelyk dit met die klassieke tegniek wat slegs simboliese waardes as invoer neem.

Acknowledgements

First of all, I would like to thank my supervisor, Professor Willem Visser, for his patience and providing me the support-technical and financial throughout my study period. You introduced me to the topic of symbolic execution and thus my interest to pursue studies on the subject. Thanks so much for always being available to discuss my ideas with you even when they were not clear at times, and understood what I meant when I was not able to express it. Without your expertise and knowledge this thesis would never have been completed. I am fortunate to work with you, you have been such a wonderful supervisor. I am greatly indebted to you.

Secondly, I would like to thank Dr. Jaco Geldenhuys and Dr. Steven Kroon for the fruitful discussions about this research we had.

I would like to thank my parents and siblings for their prayers, and the love they have shown me during the good and hard times. God bless and protect you always.

I gratefully acknowledge the financial support I received from the University of Stellenbosch and African Institute for Mathematical Sciences who jointly funded this research work.

I thank Pieter Jordaan, Jan Buys and Nyirenda for providing support, and for their invaluable ideas.

There are people who live for sharing what they have and helping, among them Caritas Nyiraneza and P. Rucogoza. For that I am forever grateful.

I would like to thank Azra Adams, Mary Nelima, Maurice Ndashimye and Steven for being there when I needed someone to lean on. It has been a pleasure of knowing each of you. May you stay blessed.

Finally, I give thanks to the Almighty God for the gift of life and the patience.

For Yezu Christu.

And for my mum Annonciata and dad Anthère.

Contents

Declaration	i
Abstract	ii
Uittreksel	iii
Acknowledgements	iv
	v
Contents	vi
List of Figures	viii
List of Tables	ix
List of Abbreviations	x
1 Introduction	1
1.1 Contributions	2
1.2 Outline of Thesis	3
2 Background and Related Work	4
2.1 Symbolic Execution	5
2.2 Java PathFinder	12
2.3 Model Counting	18
2.4 Probabilistic Symbolic Execution	23
2.5 Related Work	27
2.6 Concluding Remarks	32
3 Approach	34
3.1 Symbolic Execution Drivers	35

3.2	Classic Probabilistic Symbolic Execution	36
3.3	Extended Probabilistic Symbolic Execution	37
3.4	Implementation	43
3.5	Concluding Remarks	45
4	Results and Discussions	46
4.1	Fixed Probabilities	47
4.2	Korat Approach	49
4.3	Discussion	53
5	Conclusions and Future Work	54
5.1	Future Work	55
A	Appendix	57
A.1	Code for the Add method	57
A.2	Code for the Find method	58
A.3	Code for the Delete method	58
B	Appendix	61
B.1	Basics of Probability Theory	61
	List of References	63

List of Figures

2.1	Symbolic execution tree for code fragment 1.	8
2.2	Symbolic execution tree for the code that swaps two integers [8].	9
2.3	Lazy symbolic execution (LSE) algorithm [54].	11
2.4	A linked list with non deterministic choices [55].	12
2.5	JPF Model checking tool [2].	13
2.6	States, Transitions and Choices [2].	14
2.7	Symbolic PathFinder overview [81].	16
2.8	Trees generated for finBinaryTree(3) [18].	23
2.9	Probabilistic symbolic execution chain.	24
2.10	Probabilistic symbolic execution tree for the triangle problem.	26
2.11	Reliability analysis methodology [39].	29
3.1	Extended probabilistic symbolic execution chain.	35
3.2	Probabilistic symbolic execution tree for the code in Listing 3.3.	39
3.3	Probabilistic symbolic execution tree for the code in Listing 3.3 using Korat.	42

List of Tables

2.1	Classification and probabilities for the triangle problem [41].	27
4.1	Probability of covering locations in Binary Tree [0..9]	48
4.2	Symbolic Values: Maximum probabilities for locations in Binary Tree	50
4.3	Symbolic Structures: Maximum probabilities for locations in Binary Tree . .	50
4.4	Symbolic Values: Minimum probabilities for locations in Binary Tree	52
4.5	Symbolic Structures: Minimum probabilities for locations in Binary Tree . .	52

List of Abbreviations

HPC : Heap Path Condition

JPF : Java PathFinder

JVM : Java Virtual Machine

LattE : Lattice point Enumeration

PC : Path Condition

$P(E)$: Probability of an event E

SAT : Satisfiability

SMT : Satisfiability Modulo Theory

SPF : Symbolic PathFinder

Chapter 1

Introduction

Globally, billions of dollars are lost due to software system failure every year. For example, Toyota recalled more than 13 million vehicles worldwide due to an error in its vehicles' software that gave faulty speed readings; this failure cost Toyota an estimated of 2-5 billion US dollars [7]. Other examples include the European Space Agency's Ariane 5 Flight 501 which was destroyed 40 seconds after takeoff and a 1 billion US dollar prototype rocket self-destructed due to a bug in the on-board guidance software [37]. Despite the technological advances in languages and tools to support program development, programmers still deliver software with lots of errors [29, 6]. A way of avoiding these losses is to better understand the behaviors of a program to enable effective software testing, that will in turn ensure better system reliability.

In the software engineering field, testing software is considered as the most important method to finding and eliminating software errors. It is a very expensive activity, and a study done in 2002 by the National Institute of Standards and Technology reports that between 70% and 80% of development costs is due to testing [29, 73]. The importance of testing is growing as the impact of software errors on industry becomes more pronounced. Although testing has become a dominant method and an important part of the software development process, studies indicate that the tools used for testing the software are insufficient. Hence, the production of high quality code remains a critical issue. Different techniques and methods have been explored by various researchers. Unfortunately, the necessary level to establish the correctness of the software cannot always be guaranteed by these techniques, the tools that have been developed, provides limited support for testing in general and understanding the program's behavior.

Recent progress in software testing and verification have led to a considerable increase in the performance of the techniques for test generation, and detecting errors based on

symbolic execution [56]. The main idea behind this technique is to use symbolic values, instead of actual (concrete) values, as input values and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs (Section 2.1.1).

Nowadays, program analysis and testing based on symbolic execution have received a lot of attention, there are quite a few tools available that perform symbolic execution for programs written in modern programming languages [58, 61, 65, 81, 96]. Scaling symbolic execution remains a challenging problem especially with the analysis of programs that manipulate data structures due to issues like aliasing [11, 83].

In this thesis, we are specifically interested in analyzing programs that manipulate data structures. In particular we are interested in calculating the probability of execution behaviors. The motivation for this work is two-fold: on the one hand we would like to better understand program behavior and on the other hand we can use execution probabilities to determine software reliability. Note that we define reliability as the probability of the program not producing an error. In previous work [41] it was shown how one can calculate execution probabilities for programs that only manipulate integer variables, here we extend it to handle data structures as well.

We use Java PathFinder (JPF), a software model checker engine for the Java programming language [3] and two of its extensions: Symbolic PathFinder (SPF), an extension to Java PathFinder for performing symbolic execution, and Probabilistic Symbolic Execution (JPF-Probsym), an extension to SPF that enables the calculation of probabilities for programs with only linear integer arithmetic constraints.

To calculate execution probabilities one must be able to count the number of solutions to constraints. Here we will use the LattE [4] and Korat [18] tools to count data constraints and structures respectively. Note that counting solutions to constraints only work on the underlying assumption that values are uniformly distributed in their respective domains. This restriction can be relaxed, as shown in [39], but to simplify the exposition here we only consider uniform distributions.

1.1 Contributions

In this thesis, an existing probabilistic symbolic execution framework [41] that combines symbolic execution and model counting techniques to calculate the probability of an execution of programs for supporting linear integer arithmetic, is extended to handle data structures. We describe two approaches to handling data structures and evaluate

it on a Binary Search Tree container class. The system supports the understanding of program's behaviors, thus, enhances the software testing phase.

The contributions of this thesis are:

1. A description of how an existing probabilistic symbolic execution can be extended to allow symbolic structures as input.
2. Show two possible solutions: the first shows how we can get an approximate answer in an efficient fashion, and the second solution, gives a precise answer using the Korat tool.
3. Evaluate our approaches for the extended probabilistic symbolic execution on a Binary Search Tree example.

1.2 Outline of Thesis

This thesis is organized into five chapters and structured as follows:

- Chapter 1 serves as an introduction by describing the domain research, presenting research problem, contributions and what the thesis contains.
- Chapter 2 provides the necessary background information for the reminder of the thesis. It contains a survey of techniques that are commonly used. The concepts of symbolic execution for the linear integer arithmetic and programs with heap objects structures are provided. The fundamental notions on Java Pathfinder, Symbolic PathFinder and model counting for both integers and structures are presented. The probabilistic symbolic execution approach is described. We also discuss related research work.
- Chapter 3 presents the approaches used to extend the probabilistic symbolic execution. It shows how the existing system can be modified if we assume a fixed set of probabilities for all structural choices, which would give us an approximate answer. In addition we then show how we can use the Korat tool to count structures which will give a precise answer, but doesn't scale well to large data domains.
- Chapter 4 presents the results of experiments conducted on a Java version of a Binary Search Tree. We compare the existing probabilistic symbolic execution with the two approaches from Chapter 3.
- Chapter 5 contains the conclusions of the thesis and discusses future work.

Chapter 2

Background and Related Work

In the process of software development, effective testing is the accepted technique to find errors in the software. However, the necessary level of effort for manual test input generation is high and usually results in inadequate test cases. Various researchers have proposed automated techniques for test-input generation [9], one such technique is symbolic execution.

Early symbolic execution [56] has been proposed to manipulate the programs with primitive data types, such as integers, and researchers have recently focused on how to handle arrays and reference types [54, 11]. Other techniques have been introduced for automated reasoning technologies, one of them called model counting [43, 39, 41], is frequently used for solving artificial intelligence problems, such as probabilistic reasoning, which includes Bayesian net reasoning [87]. In this thesis, the use of model counting in software engineering for supporting the testing phase, is explored.

There is a plethora of research on symbolic execution and model counting techniques. However, the use of both techniques for automatic testing and verification of programs is an emerging field. Our investigation focused on the presentation of the available techniques related to the use of symbolic execution, and model counting for both primitive types (e.g., integers) and reference types (e.g., structures) in testing and verification of Java programs. We therefore do not expend much effort in describing model counting in detail; rather, we identify and discuss specific model counting tools chosen to be used in this thesis. The reader interested in model counting techniques and their applications may refer to [44].

The chapter begins with a general description of symbolic execution in Section 2.1, symbolic execution for integers in Subsection 2.1.1 and for programs with heap objects via lazy initialization in Subsection 2.1.2. Section 2.2 gives a background on Java PathFinder

and Symbolic PathFinder. The model counting concepts for integers and structures are described in Section 2.3. Section 2.4 provides the goals of probabilistic symbolic execution and techniques used to realize these goals. Section 2.5 discusses the techniques and studies most closely related to this thesis. We conclude the chapter with the concluding remarks in Section 2.6.

2.1 Symbolic Execution

In the mid 1970's, King [56] and Clarke [27] introduced symbolic execution, a program analysis technique that performs execution of a program on symbolic values rather than concrete data inputs. This technique was mainly used for program testing and debugging. Even though this technique was explored by various researchers to accomplish different kinds of analyses since its beginning, it was only during the last decade that the technique started to realize its powerful analysis potential in the context of exposing errors in software, generating high-coverage test cases and enabling the understanding of the behaviors of programs [20, 22, 26, 41, 45, 54, 91]. This is due to the recent dramatic growth of algorithmic advances and to the increased availability of powerful constraint solving technology and computational resources [21].

One basic advantage of symbolic execution over concrete execution (e.g., traditional testing) is that symbolic execution can reason about unknown values represented by symbols (or symbolic values) (e.g., α, β, x, y etc.) instead of concrete values (e.g., integers) [36].

A number of tools for symbolic execution are currently available in public domain [8]. For Java, available tools include Symbolic PathFinder [80], JFuzz [51], and LCT [52]. For C, tools that are available include Klee [20], S2E [24], and Crest [50].

2.1.1 Symbolic Execution for Integers

Symbolic execution [56, 76] is a popular static analysis technique used in software testing to explore as many different program paths as possible in a given amount of time, and for each path, it generates a set of concrete input values exercising it with the aim of checking the presence of several kinds of errors, including undetected exceptions and assertion violations [22].

The main idea behind symbolic execution technique is to execute the code of program using symbolic values as inputs in place of concrete values and represent the values of program variables as symbolic expressions over the symbolic values. As a result, the

output values computed by programs are expressed as a function of symbolic inputs [81, 22].

To ease understanding, in the continuing text, English letters are used to represent the variables, Greek letters are used to represent the symbolic values and the symbol " \leftarrow " indicates assignments of values to variables.

In symbolic execution [56], a program may be represented by a control flow graph, a directed graph that contains many or an infinite number of paths. It explores the execution of a program tree where a node represents a symbolic state and the transitions between states are represented by the arcs or edges. The program that is executed symbolically comprises three states [56]:

1. A path condition (a condition on the inputs symbols such that if a path is feasible its path condition is satisfiable).
2. Symbolic values of program variables.
3. A program counter (points to the current statement of the method being executed. In other words, it indicates the next statement to be executed).

Definition 2.1.1 *Definition (Path Constraint) [31]. The path constraint (PC) of a program path p is a boolean formula over the symbolic inputs, this is a logical conjunction of conjuncts that the program inputs must satisfy for an execution to follow that path p .*

The path associated with a path condition can be executed concretely using input values that satisfy the constraints in the path condition. The paths generated during the symbolic execution of a program are characterized by a symbolic execution tree [81]. To illustrate the idea behind symbolic execution, we consider the algorithm 1 and the example which illustrates it in Listing 2.1.

Algorithm 1 *SymbolicExecute*(l, ϕ, m, p) [41].

```

while  $\neg \text{branch}(l)$  do
   $m \leftarrow m\langle v, e \rangle$ 
   $l \leftarrow \text{next}(l)$ 
end while
 $c \leftarrow m[\text{cond}(l)]$ 
if  $\text{SAT}(\phi \wedge c)$  then
  SymbolicExecute( $\text{target}(l), \phi \wedge c, m$ )
end if
if  $\text{SAT}(\phi \wedge \neg c)$  then
  probSymbolicExecute( $\text{next}(l), \phi \wedge \neg c, m$ )
end if

```

The symbolic execution algorithm 1 adapted from [41] outlines the basic elements of symbolic execution. It contains initial location of the program represented by l , the path condition which is true, and an initial map represented by m . It operates by decomposing symbolic executions into different locations that are placed between branch statements. These are mainly represented by $\text{branch}(l)$ whose condition is represented by $\text{cond}(l)$. Beside this, it also contains non-branching statements whose form looks like $v = e$.

When all non-branch statements are processed, their outcomes are examined. With positive branch outcome whose formula is found to be satisfiable, will become a new path condition and the next branch of the code to be processed is the target location.

With a negative branch outcome, the process remains the same with an exception of negating the branch condition, and the next branch to be processed starts at the next location. As an illustrative example, consider the code fragment 1 in Listing 2.1, that increments or decrements the value of an integer, when the initial value of x is greater than zero or less than or equal to 0. The statements are referenced by their line numbers.

```

1 int example (int x)
2 {
3   if (x > 0)
4     x++; // S1
5   else
6     x--; // S2
7   return x;
8 }

```

Listing 2.1: Code fragment 1.

At every conditional statement *if S1 else S2*, the path condition is updated. To symbolically execute this program, its behavior is taken into consideration and analysed, when the input variable x contains a symbolic value α , the method *example* is invoked, and

takes a single argument x . If x is greater than 0, the value of x will be incremented otherwise i.e., if it is less than or equal to 0, it will be decremented.

At the first statement, symbolic execution considers two constraints: $(\alpha > 0)$ and $\neg(\alpha > 0)$ in other words, $(\alpha \leq 0)$.

When $(\alpha > 0)$, the value of x is incremented at statement 4, and then the value $\alpha + 1$ is returned at statement 7.

When $\neg(\alpha > 0)$, the value of x is decremented at statement 6, and then the value of $\alpha - 1$ is returned at statement 7.

The symbolic execution tree which represents the execution paths followed during the symbolic execution of the given code fragment 1, is shown in Figure 2.1.

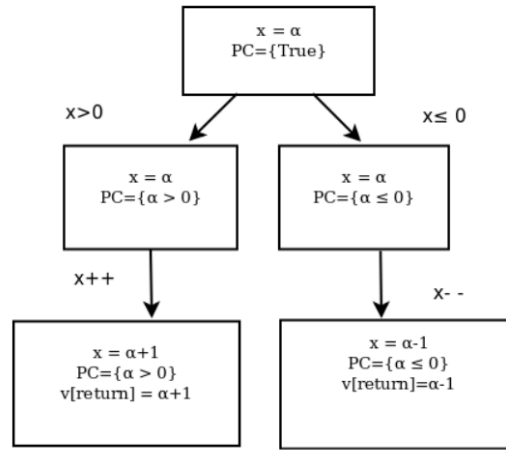


Figure 2.1: Symbolic execution tree for code fragment 1.

Symbolic execution normally uses that fact that the path is either satisfiable or unsatisfiable [81]. The determination of a satisfiability or unsatisfiability of the path conditions is performed by various decision procedures tools such as, CVC3 [12], Choco [1] and Z3 [31]. These tools vary in the types of constraints they can solve. For example, CVC3 is used for solving real and integer linear arithmetic, and also the bit vectors operations. Choco is implemented in Java and used to solve linear/non-linear integer/real constraints. Z3 is implemented in C++ and used in various software verification and analysis applications [31].

Consider an example taken from [8], Listing 2.2 shows a program which swaps integer values for variables x and y , when the initial value of x is greater than y . Its corresponding symbolic execution tree is shown in Figure 2.2.

```

1  int x, y;
2  read x, y;
3  if (x>y) {
4      x = x+y;
5      y = x-y;
6      x = x-y;
7      if (x-y>0)
8          assert false;
9  }

```

Listing 2.2: Code that swaps two integers [8].

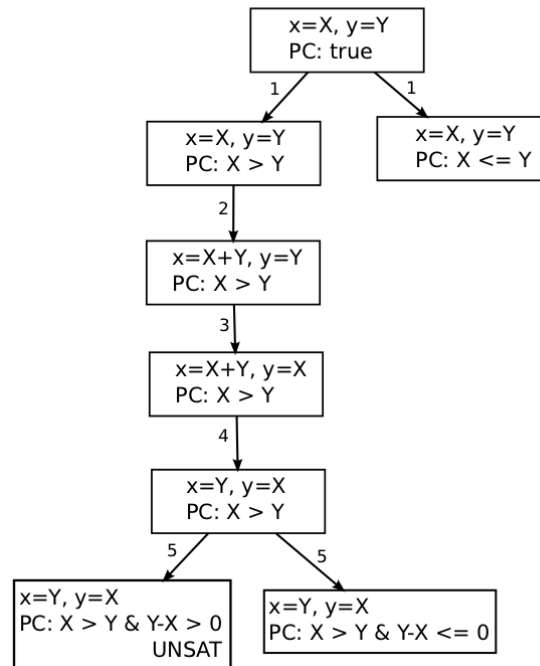


Figure 2.2: Symbolic execution tree for the code that swaps two integers [8].

The process starts with the path condition which is true, which means that before the execution of the *if* statement at line 3 where x is greater than y , the PC is initialized to true. For any program input, the symbolic values X and Y are given to x and y . Then after the execution of the *if* statements at the lines 3 and 7, PC is updated appropriately. After the execution of the first statement in line 3, there are two possible alternative inputs that are found to be satisfiable with the "if" statement i.e., *then* and *else*.

On one hand, there is a set of constraints $X > Y \ \& \ Y - X \leq 0$ for which the program has inputs which allow the swapping of the integers, this happens when $x=2$ and $y=1$. On the other hand, the path (1,2,3,4,5,6) having $X > Y \ \& \ Y - X > 0$ as path constraints, is found to be unsatisfiable. This means that the program does not have any inputs for

which it can take the infeasible path. Therefore, code is considered to be unreachable and the symbolic execution backtracks, see the Figure 2.2.

2.1.2 Symbolic Execution of Programs with Heap Objects-Lazy Initialization

In this section, details of how symbolic execution for programs with heap objects operate, are provided. The technique used is called lazy initialization and more details can be found in [54, 42].

Lazy Initialization [54] is a technique that delays the creation of an object until the first time it is needed. It has been used with the symbolic execution technique to handle the programs that have the heap object structures and arrays as inputs [54]. This has potentially contributed to the path explosion problems since it was observed that the manipulation of an object oriented program is notoriously hard due to issues of aliasing [54, 83]. Further work done with the aim of manipulating the heap structures and arrays using subsumption checking was performed by Anand et al. [11].

The main idea behind the LSE algorithm shown in Figure 2.3, implemented in SPF (Section 2.2.2), is that it starts the symbolic execution of a procedure on un-initialized input and uses lazy initialization to assign values to these inputs. Thus, lazy initialization provides a method for systematically exploring heap configurations in a programming language like Java that enforces the manipulation of the heap. For a given program, lazy initialization works in the same way as symbolic execution i.e., it starts with no knowledge of the heap structure and symbolically executes the program to discover and initialize the heap structure, and the unknown object values are represented by special symbols.

With the lazy symbolic execution algorithm (LSE), when a program executes and accesses an object field, it initializes the values to the field on demand. The LSE first checks whether the field is initialized. If the field is not yet initialized, then the algorithm checks its type, i.e. if the field type is scalar, then a fresh symbol is created for that scalar value which refers to an object. For an un-initialized reference field, the algorithm explores all possible options by non-deterministically initializing the field and choosing among the following values for the reference as presented in the Figure 2.3.

```

if ( f is uninitialized ) {
  if ( f is reference field of type T ) {
    nondeterministically initialize f to
    1. null
    2. a new object of class T (with uninitialized field values)
    3. an object created during a prior initialization of a field of type T
    if ( method precondition is violated )
      backtrack();
  }
  if ( f is primitive (or string) field )
    initialize f to a new symbolic value of appropriate type
}

```

Figure 2.3: Lazy symbolic execution (LSE) algorithm [54].

Note that the second case may lead the lazy initialization to continue expanding the heap and not to terminate because of the possibility of creating more choices, this can be overcome by limiting the depth of a path. During the initialization of a reference field, lazy symbolic execution also checks for the method's precondition with the aim of handling its violation. With the primitive fields, when a branching condition is evaluated, the lazy initialization algorithm non-deterministically adds the condition or its negation to its path condition and checks whether the path condition is satisfiable or not. This satisfiability checking is performed with the aid of decision procedure as previously mentioned. In case the path condition is found to be infeasible, the current execution terminates, that is to say, the algorithm backtracks. In addition, LSE supports the fundamental foundation necessary for carrying out symbolic execution on programs in order to manipulate dynamically allocated data structures. When the field is un-initialized and also is a non reference type field, LSE follows traditional symbolic execution since it is developed in the context of sequential programs which contain a fixed number of program variables having the primitive types such as integers. As an example, consider the code shown in Listing 2.3 which implements a linked list.

```

1 public class ListNode {
2   private ListNode next;
3   private Object value;
4   public void add(Object k) {
5     if (next == null) {
6       ListNode n = new ListNode ();
7       n.value = k;
8       this.next = n;
9     }
10    else
11      next.add(k);
12  }

```

Listing 2.3: A linked list example [55].

We illustrate the lazy initialization algorithm on the linked list program represented in Listing 2.3, the program presents `LinkedList` that implements a linked lists. The object's value and `LinkedList` next represents, the node's integer value and a reference to the next node in the list respectively. For a given object reference `k`, the LSE starts to operate in line 5 at the "if" statement, i.e., the first time the field is accessed, the linked list is extended. Lazy initialization chooses non-deterministically the choices among all possibilities for the field of that object [55], and the linked node `n` will have 4 choices created earlier normally called alias choices, besides this, it will also have null and new choices, as described in 2.4.

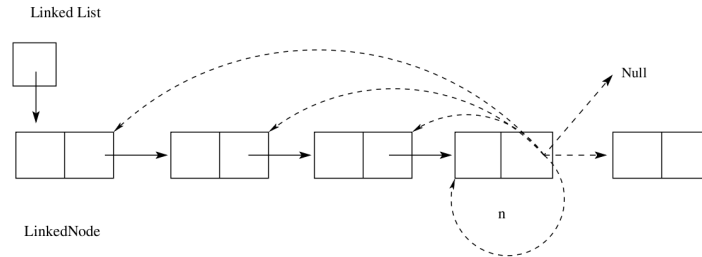


Figure 2.4: A linked list with non deterministic choices [55].

Figure 2.4 illustrates a simple linked list with non-deterministic choices performed by lazy initialization, as explained in Figure 2.3.

It has been reported that the decision procedure used to solve path conditions and check their satisfiability, are utilized only for scalar values and not for heaps [35]. Since our focus is based on handling heap object structures by using a lazy initialization algorithm, the decision procedures are not used. From a different point of view, lazy initialization can be considered as a decision procedure for object structures with case splitting on possible aliasing scenarios [35].

2.2 Java PathFinder

Java PathFinder(JPF) [3, 92] is an open-source implementation of the Java Virtual Machine for verifying Java bytecode, developed at National Aeronautics and Space Administration (NASA) Ames Research Center. This is an explicit state model checker for Java bytecode, and contains a core package, i.e., JPF-Core with other extensions such as, JPF-Awt, Symbolic PathFinder (SPF), and JPF-Probsym. We are specifically interested in the probabilistic symbolic execution extension that has been created for it: JPF-Probsym. Figure 2.5 presents the components of JPF, namely:

- A model, a system under test.
- A model checker, JPF is itself a virtual machine.
- A specification, a JPF configuration.

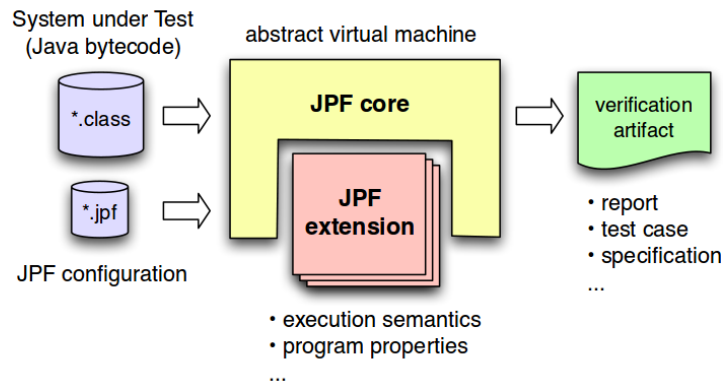


Figure 2.5: JPF Model checking tool [2].

JPF is implemented in Java as a special Java Virtual Machine (JVM) that runs on top of the host JVM. Therefore, it handles all standard Java features and in addition allows for non-deterministic choices written as annotations, these annotations are added by method calls to class *Verify* [88].

The inputs to JPF are: the class files (Java bytecode) for a system under test and a set of configuration text files which specify the desired JPF execution mode, program properties to verify, and artifacts to generate. The verification artifacts produced are usually reported in various formats [81].

It takes as input a Java program (and an optional bound on the length of program execution) and explores all executions (up to a chosen depth bound) that the program can have due to different non deterministic choices, and generates as output executions that violates given properties, test inputs for the given program or state-space exploration [92]. The class files of the Java program are analyzed by interpreting the Java bytecodes in a custom-made Virtual Machine. It also implements a (default) concrete execution semantics that is based on a stack machine model, according to the Java Virtual Machine specification [59].

JPF's core is a state exploring JVM which can examine alternative paths in a Java program (for example, via backtracking) by trying to provide all non-deterministic choices, including thread scheduling order. It explores all executions that a given Java program can

have and implements a backtrackable Java Virtual Machine to support non-deterministic choices e.g., in thread interleavings and provides the control over thread scheduling.

The main difference between JPF and a regular JVM is that JPF can quickly backtrack the program execution by restoring the previous states on a path encountered during the execution. Backtracking allows the exploration of different executions from the same state. To perform the backtracking faster, JPF uses a special representation of states and executes program bytecodes by modifying this representation. Specialized JVM explores all possible execution paths of a Java program. The core of JPF is a special Java virtual machine that supports backtracking, state matching, and non-determinism of both data and scheduling decisions or choices.

2.2.1 Choice Generators

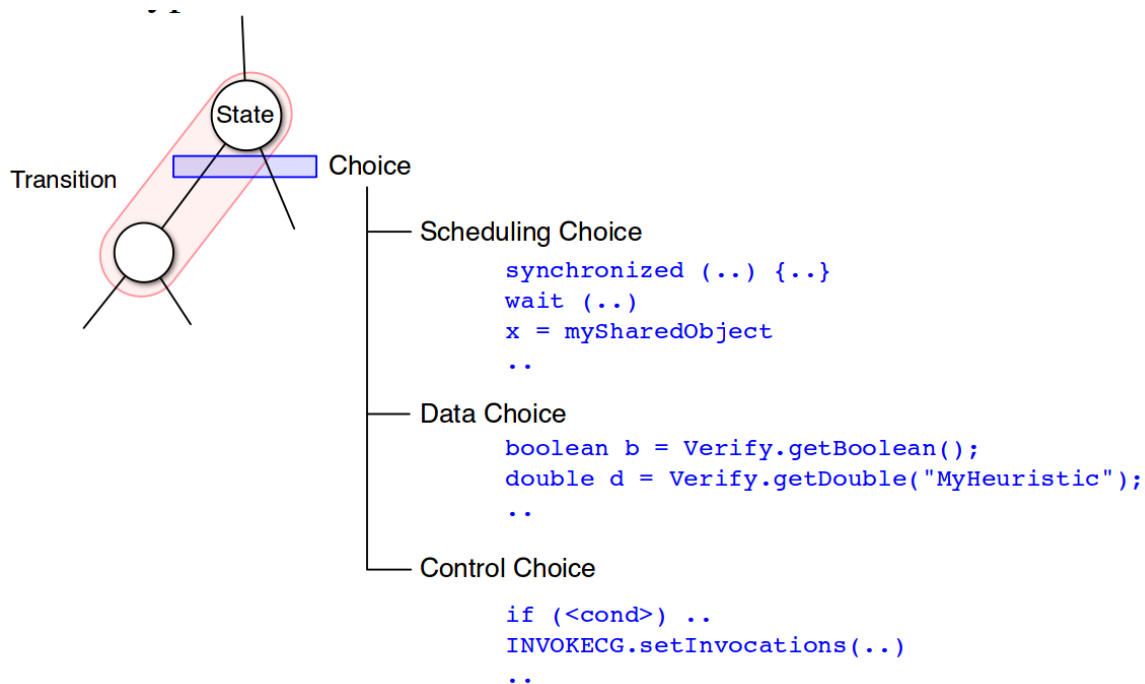


Figure 2.6: States, Transitions and Choices [2].

In order to explore the state space, JPF uses choice generators [2]. This is defined as the mechanism used by JPF for systematically exploring the state space. It corresponds to non-deterministic choices made during execution, and is often generated by instrumentation in the source code being explored [89]. There are various types of choices, namely, scheduling, data, control, and user defined choices. A new transition is started with one of the type of choices and extend until the next as can be seen in Figure 2.6, when there are

unprocessed choices, backtracking moves up to the next choice generator. Data choices can often be created programmatically by using the `verify` package. For instance, if one needs to perform the verification of any program with input values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, then the specification can be done as, `Verify.getInt (1,10)`.

Note that the non deterministic choice mode happens when at a given state, more than a single transition is enabled. For choosing the threads, JPF provides `ThreadChoiceGenerator` and all classes related to data and scheduling choice are kept in the `gov.nasa.jpf.jvm.choice` package. For symbolic execution (Section 2.1.1), a `PC ChoiceGenerator` is used to consider whether constraints forming a PC are feasible or not.

In [74], it was reported that JPF constructs the program state space on-the-fly and at the end of each transition. All the properties are checked (which may be built-in properties such as race conditions and deadlocks). A transition is a sequence of bytecode instructions executed by a single thread; only the first instruction in the sequence represents the non-deterministic choice. At every transition boundary, JPF saves the current JVM state in a serialized form for the purpose of backtracking and state matching. The complete JVM state includes all heap objects, stacks of all threads and all static data [74]. During the program's execution it takes a Java program as input and explores all the executions that the program may have due to different non-deterministic choices. It represents the JVM state of the Java program being checked and performs bytecodes execution, backtracking for storing and restoring state such that it backtracks the execution during the state space exploration, and state comparison for detecting cycles in the state space.

JPF has different extensions as previously mentioned. These are considerably efficient for automatic test generation and utilized during the verification of programs. Depending on the type of analysis performed, they are useful in the exploration of program paths, detection of errors as well as the creation of test drivers. In this work, however, only two extensions are used:

- JPF-Symbc (SPF), an extension to JPF for performing symbolic execution.
- JPF-Probsym, an extension to SPF for performing probabilistic symbolic execution.

2.2.2 Symbolic PathFinder (SPF)

Symbolic PathFinder (SPF) [2] is an extension project in JPF and it is available as the project JPF-Symbc from the JPF distribution. This SPF extension allows for the

symbolic execution of Java bytecode, including LSE for reference types (see Section 2.1.2). More details on JPF-Symbc are available online ¹.

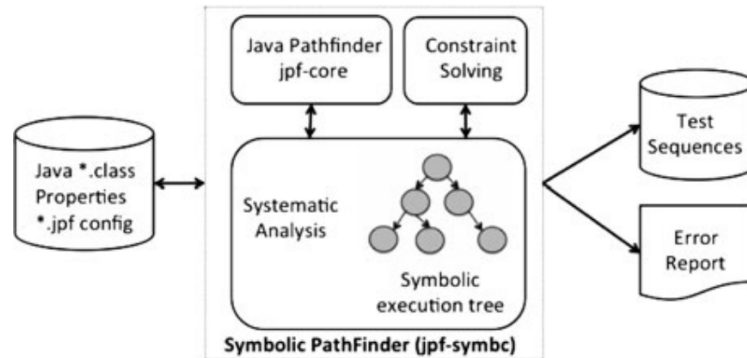


Figure 2.7: Symbolic PathFinder overview [81].

SPF relies on the Java PathFinder model checker (*JPF – core*) to systematically explore the different symbolic execution paths, as well as different thread interleavings. Furthermore, SPF utilizes JPF’s built-in strategies for state space exploration, such as depth-first search or breadth-first search [81]. The limitation of possible infinite search space can occur with the symbolic execution of programs with loops, is overcome by limiting the depth of a path. As input, SPF requires [81]:

- the class files of an executable program,
- a configuration file specifying which methods in the program should be executed symbolically,
- properties being verified or a test coverage criteria to obtain a test suite.

SPF combines symbolic execution, constraints solving and model checking for test case generation and error detection. One main application is to automatically generate a set of test inputs that achieves high code coverage (e.g., path coverage) [79].

There are a few tools available that perform symbolic execution for programs written in modern programming languages [91, 58, 96]. What distinguishes SPF from these tools is its ability to handle complex symbolic inputs and multithreading, and its extensibility due to several works done and many applications built on top of SPF [81].

¹ <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc>

The state of a program which is executed symbolically contains symbolic values of the program's variables, the program counter, and a path condition (PC), this is the boolean formula which presents the constraints that should be satisfied by the symbolic values during the program's execution. A symbolic state of any program P has a symbolic heap configuration H and a path condition PC . Besides this, it also includes the program counter and thread scheduling information [79].

SPF implements symbolic execution with the aid of a non standard bytecode interpretation. It integrates symbolic execution [56, 25] with model checking [92] to perform automated generation of test cases and to check properties of the code during test case generation. Whenever a path condition is updated, it is checked for satisfiability or un-satisfiability using an appropriate decision procedure. If the path condition is not satisfied, the model checker backtracks. When the satisfiability of the path condition cannot be determined i.e. when it is undecidable, the model checker still backtracks. Therefore, in this case, the feasible program behaviors can only be explored by the model checker.

The implementation of SPF is performed through the change of JPF's standard bytecode interpretation which performs concrete symbolic execution as we indicated that the symbolic execution tracks symbolic values rather than concrete values. These concrete values can be integers, floats and so on. This is done to explore the state space of the byte codes which are extended to allow the variables to be represented by symbolic values and expressions, generating path condition in case of the execution of conditional byte codes and the storage of the symbolic execution by using variable attributes. This storage is achieved by assigning symbolic attributes to variables and fields. The SPF's bytecode is a true extension of the standard concrete bytecodes, both concrete values and symbolic values can be used during the same execution [89]. When a conditional bytecode (e.g. those compiled from "if" statements, "switch" statements, etc.) is executed in SPF, execution branches to explore the result of the bytecode and are evaluated to *"true"* or *"false"*.

In the generation of choices, the Path Condition Choice Generator is used to non-deterministically choose which branch to explore. By default, two choices, *"true"* and *"false"*, are generated. Each choice generated is associated with a path condition; the bytecode's condition; if *"true"* and the negation of the bytecode's condition if *"false"*. When a choice is explored, the bytecode evaluates this choice and the associated path condition is appended to the PC. During branching execution, the satisfiability of the path condition is checked using off-the-shelf constraint solvers. If the PC is satisfiable, JPF continues along the associated path; otherwise, JPF backtracks. To handle uninitialised inputs to the system under verification, SPF uses lazy initialisation as described

in [54] and generates the heap path constraints. The approach is used for finding counter-examples to safety properties and for generating tests. For every counter-example, the model checker reports the input heap configuration (encoding constraints on reference fields), the numeric path condition (and a satisfying solution), and thread scheduling, which can be utilized to reproduce the error [60].

Figure 2.7 illustrates the SPF’s components; the non-deterministic Java program is considered as an input whose source code is instrumented to facilitate the manipulation of the formula that describes the path conditions. This instrumentation enables JPF to perform symbolic execution. The model checker explores the symbolic state space which contains a path condition, a heap configuration and a thread scheduling. On any occasion a path condition is updated, an appropriate decision procedure can be used to check whether it is satisfiable or not. When the path condition is found to be un-satisfiable, in such case, the model checker backtracks. The testing coverage criterion is now checked by the model checking which in return produces a counter-example. Here, input variables are allowed to be symbolic and all constraints that compose a counter-example are expressed in terms of inputs.

2.3 Model Counting

We introduce some aspects of model counting relevant to our study. We start by defining what model counting is, then, we present an example that illustrates it.

Definition 2.3.1 *Model counting or # SAT [41, 44] is the problem of determining the number of solutions of a given formula, i.e. the number of distinct truth assignments to variables for which the formula evaluates to true.*

2.3.1 Model Counting for Integers

Model counting requires the solver to be cognisant of all solutions in the search space. Thus, solving a counting problem is at least as hard as solving the satisfiability problem [41]. While different categories of model counting techniques can be explored, in this thesis, the LattE [4] and Korat [18] tools are used as model counters. As an example, consider the code fragment 2 from Listing 2.4.

```

1 void foo(int x){
2   if (x > 5)
3     print x;
4   else
5     print "true";
6 }

```

Listing 2.4: Code fragment 2.

Questions: How do we go about calculating the probabilities for the path conditions ? in other words, what is the probability of getting the value of x and that for getting the message "true"? Or which path condition is more likely to be executed than other?

Symbolic execution of the code fragment 2 in Listing 2.4 explores the following two path conditions:

Path A: $[x > 5]$ $1 \rightarrow 2 \rightarrow 3 \rightarrow 8$.

Path B: $[x \leq 5]$ $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8$.

Assume the input domain of variable x is $\{0, 1, 2, \dots, 9\}$, we may use the constraints to check this set as $x \geq 0 \wedge x \leq 9$. There are 10 different input values and two path conditions. The paths can be explored in various ways; depth first order is the simplest and most commonly [41].

With the code fragment 2 presented in Listing 2.4, it is clear that for $(x > 5)$ given that $x = \{0, 1, \dots, 9\}$, there are 4 numbers of solutions for the path A, these are: $\{6, 7, 8, 9\}$. Thus, the probability of getting the value of x will be $P(A) = \frac{4}{10}$.

To calculate the probability for getting the "True" message, for "else" statements, i.e for $x \leq 5$, there are 6 number of solutions for the path B, these are: $(0, 1, 2, 3, 4, 5)$. Thus, the probability of getting the message "true", would be $P(B) = \frac{6}{10}$. Therefore, the path condition B is likely executed than A.

Model counting [44] presented challenges for the researchers and poses several new research questions. The problem of counting solutions has been studied and many attempts for finding its solution have been made by various researchers. Model counting arose from the satisfiability (SAT) problem [40, 64]. As mentioned in [44], the accurate algorithms for solving this problem will have a significant impact on many application areas that are naturally beyond SAT. Model counting is frequently used for Artificial Intelligence problems such as bounded-length adversarial and contingency planning, and probabilistic reasoning, including Bayesian net reasoning [87, 85] where recently, a number of different techniques to model counting have been presented. Its potential in solving software

engineering problems is to be exploited. An interested reader of model counting and its application can refer to [44].

2.3.2 Model Counting for Structures

We start by defining structures and their validity. This allows us to precisely explain the problem of solving structures, counting them and present their correctness requirements.

2.3.2.1 Definitions

Definition 1 (Structure) [66]: Structures are defined as rooted (object) graphs, where nodes represent objects and edges represent reference fields. Let O be a set of objects whose fields form a set F . Each object has a field that represents its class.

Definition 2 (Validity) [66]. Let γ be a structure for a predicate π , and let $\pi(\gamma)$ be the results for the execution of the predicate π , which in return, produces the structure γ . We say that γ is valid if and only if $\pi(\gamma) = \text{true}$, and γ is invalid if and only if $\pi(\gamma) = \text{false}$. We also say that a valid satisfies the predicate.

As stated in Section 2.3, the LattE model counter mentioned can be applied to count the integers, with the structures, we decided to use Korat as the counting procedure. This is explored to support the manipulation of data structures by generating constraints.

Korat [18] is a well supported framework for constraint-based generation of structurally complex test inputs for Java programs. It generates structurally complex inputs by solving imperative predicates, where an imperative predicate is a piece of code that takes an input, which we call a structure, and determines its validity [66].

Korat generates all test inputs structures (within the bounds) from the structure space that satisfy the constraints and provides the specifications based on the testing and counting of input data structures. In general, it requires :

1. An imperative predicate that specifies the desired structural constraints.
2. A finitization that bounds the desired test input size.

2.3.3 Finitization

Korat uses what is called finitization or scope. This refers to the set of bounds that limits the size of the structures, it also serves by generating a finite state space for the method

predicates of the given structure and determining the set of classes for the inputs. In other words, each finitization specifies the bounds for the number of objects in the structure and possible values for the fields of these objects. It is up to the user to choose the values for the field domains in the finitization [66].

Given a bound on the input size, called finitization or scope, Korat automatically generates all predicate inputs for which the predicate returns true. The predicates are written in a boolean method called `repOK` [18]. Previous research has shown how the use of Korat for reliability analysis of software in counting structures can be performed [39]; in this work, Korat is used as a model counting procedure. Korat performs a systematic search of the predicate's input space. A Java predicate is used to explore their space and enumerate all solutions (inputs) for which the predicate returns true. Given a data structure with a formal specification for a method, Korat performs efficient generation and counts the input data structures that satisfy complex predicates, and in return represents properties of the desired inputs. It uses two methods; (1) precondition method which generates all test cases for a given size. (2) postcondition method that is considered as a test oracle used for checking the correctness of each output.

In the process of the generation of test inputs, Korat constructs a Java predicate (i.e., a method that returns a boolean expression). After the generation of the predicate and a set of bounds on the size of its inputs (called finitization), Korat generates all nonisomorphic valid structures within the given scope, i.e., all test inputs up to the given size bound. For example, Korat generates five non-isomorphic trees of 3 nodes as shown in Figure 2.8.

In the case of graphs, Korat does not actually generate all valid object graphs but only non-isomorphic object graphs. Two object graphs are isomorphic if they differ only in the identity of the objects in the graphs [18, 66, 68]: isomorphic object graphs have the same branching structure (same shape) and the same values for primitive fields.

As an illustration, consider a simple data structure in Listing 2.5, the Binary Tree whose Java source code is adapted from [18]. It contains the Java type specification of Binary Tree and Node as a Java class. It also includes the `repOk()` method, which is the Java predicate used as the precondition method. Listing 2.6 presents its finitization.

Listing 2.5: Binary Tree and its representation predicate repOK [18].

```

1 class BinaryTree {
2   private Node root; // root node
3   private int size; // number of nodes in the tree
4   static class Node {
5     private Node left; // left child
6     private Node right; // right child
7   }
8   public boolean repOk() {
9     if (root == null) return size == 0;
10    Set visited = new HashSet();
11    visited.add(root);
12    LinkedList workList = new LinkedList();
13    workList.add(root);
14    while (!workList.isEmpty()) {
15      Node current = (Node)workList.removeFirst();
16      if (current.left != null) {
17        // checks that tree has no cycle
18        if (!visited.add(current.left))
19          return false;
20        workList.add(current.left);
21      }
22      if (current.right != null) {
23        // checks that tree has no cycle
24        if (!visited.add(current.right))
25          return false;
26        workList.add(current.right);
27      }
28    }
29    if (visited.size() != size)
30      return false;
31    return true;
32  }
33 }

```

Listing 2.6: Finitization for Binary Tree [18].

```

1 public static Finitization finBinaryTree(int NUM_Node) {
2   Finitization f = new Finitization(BinaryTree.class);
3   ObjSet nodes = f.createObjectSet("Node", NUM_Node);
4   // #Node = NUM_Node
5   nodes.add(null);
6   f.set("root", nodes); // root in null + Node
7   f.set("size", NUM_Node); // size = NUM_Node
8   f.set("Node.left", nodes); // Node.left in null + Node
9   f.set("Node.right", nodes); // Node.right in null + Node
10  return f;
11 }

```

The predicate inputs have objects from various classes which form a class domain. These classes contain the objects that the field may present. Listing 2.5 presents the given predicate's inputs and the method that creates *Num_Node* objects.

To illustrate the use of Korat, consider an example of a binary tree with the invocation of Korat (it invokes repOK). Korat allocates the objects by assigning one binary tree's input object to two fields i.e. root and size. There are three node objects, namely, N0, N1, and N2 respectively. Each of them has a left and a right node as fields.

Each object of the class BinaryTree represents a tree. The size field contains the number of nodes in the tree. Objects of the inner class Node represent nodes of the trees. The method repOk first checks if the tree is empty. If not, repOk traverses all nodes reachable from root, keeping track of the visited nodes with the aim of detecting cycles.

To generate trees that have a given number of nodes, Korat tool uses the finitization shown in Listing 2.6. Each reference field in the tree is either null or points to one of the Node objects, the parameter NUM Node presents the bound on number of nodes in the tree [18].

The predicate inputs for Binary tree is composed of 8 fields, hence, the state space of inputs is composed of all possible assignments to all fields and each of the fields contains a value from its corresponding field domain [18]. Korat accomplishes a search over all assignments determined by the finitization [18]. When considering the example presented in Listing 2.5, all nonisomorphic trees generated by Korat are shown in Figure 2.8.

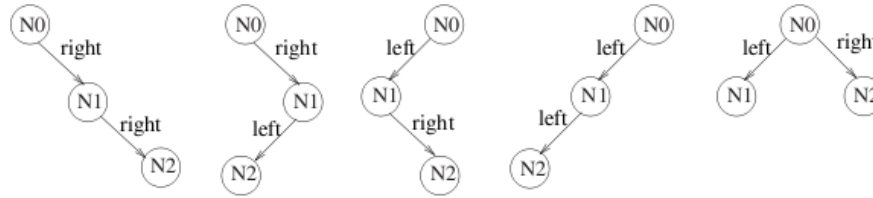


Figure 2.8: Trees generated for finBinaryTree(3) [18].

2.4 Probabilistic Symbolic Execution

Probabilistic symbolic execution [41] is a technique that combines symbolic execution and model counting techniques for calculating path condition probabilities of a Java program.

Calculating the probability for a path condition, requires the counting of the number of solutions for that path condition and the counting of the total number of values which compose the input domain size. Thus, the probabilities will simply be calculated as the number of solutions to a path condition divided by the total number of values of the

input domain size. This works for counting the solutions when the inputs are uniformly distributed within their domain [41].

The implementation of probabilistic symbolic execution was performed in two main steps as illustrated in Figure 2.9. This illustration was developed based on the method outlined in [41].

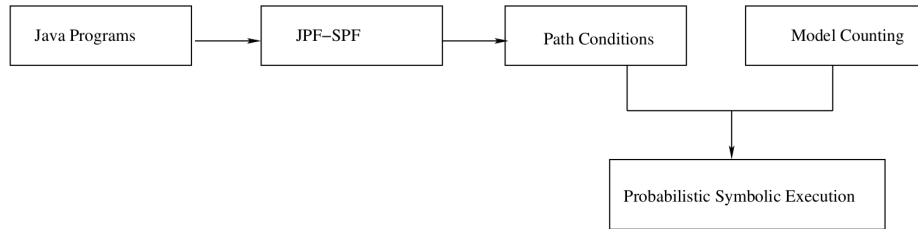


Figure 2.9: Probabilistic symbolic execution chain.

The first step starts with the input of the process that is a Java program which is symbolically executed by SPF whose output is a set of path conditions. The second step is the use of model counting for counting the solutions to a path condition yielding the results for path conditions probabilities. To count the solutions for a path condition, LatTE [4] is used. This is commonly used in practice for computing volumes for both real and integral of convex polytopes [41, 62], as well as integrating functions over those [63]. The former can be used to compute path probabilities when input variables are drawn uniformly from their type’s domain, or if a probability mass function is available for integral variables. The latter is used when a probability density function is available for real-valued variables [41].

In [41] optimizations based on path conditions slicing as well as count memoization are employed to the above method to reduce the cost of calculating the probabilities to a path condition. The path condition slicing enables one to reduce the size of path condition (i.e., obtain minimal path condition) to be checked for satisfiability. The algorithm 2 slices the PC, ϕ , with respect to the branch condition c this was performed with the aim of reducing both the size of the constraint and the number of variables involved, which leads to faster model counting. Slicing presents the opportunity of computing a small formula and therefore memoization can reuse again these computations which come from different parts of a symbolic execution tree [41]. It has been reported that when the complete path condition is very large, then, one can first slice the path condition to only obtain the part that is used to determine if the current condition is feasible [81]. However, that this also means one can only calculate conditional probabilities that just state what

the odds are of taking the current branch (without considering the previous branches). The complete path probability is calculated by multiplying all conditional probabilities along the path [81], this can be described by the algorithm 2.

Algorithm 2 *probSymbolicExecute*(l, ϕ, m, p) [41].

```

while  $\neg \text{branch}(l)$  do
     $m \leftarrow m\langle v, e \rangle$ 
     $l \leftarrow \text{next}(l)$ 
end while
 $c \leftarrow m[\text{cond}(l)]$ 
 $\phi' \leftarrow \text{slice}(\phi, c)$ 
 $p_c \leftarrow \text{prob}(\phi' \wedge c) / \text{prob}(\phi')$ 
if  $\text{SAT}(\phi' \wedge c)$  then
    probSymbolicExecute( $\text{target}(l), \phi \wedge c, m, p * p_c$ )
end if
if  $\text{SAT}(\phi' \wedge \neg c)$  then
    probSymbolicExecute( $\text{next}(l), \phi \wedge \neg c, m, p * (1 - p_c)$ )
end if

```

2.4.1 Example

The piece of code presented in Listing 2.7, classifies the type triangle when it has three side lengths.

Listing 2.7: Solution for Myers's triangle problem [41].

```

1 int classify (int a, int b, int c) {
2   if (a <=0 || b <=0 || c <=0) return 4;
3   int type =0;
4   if (a==b) type +=1;
5   if (a==c) type +=2;
6   if (b==c) type +=3;
7   if ( type = =0) {
8     if (a+b <=c || b+c <=a || a+c<=b) type =4;
9     else type =1;
10    return type ;
11  }
12  if (type >3) type =3;
13  else if ( type ==1 && a+b>c) type =2;
14  else if ( type ==2 && a+c>b) type =2;
15  else if ( type ==3 && b+c>a) type =2;
16  else type =4;
17  return type ;
18 }

```

Assume that $a, b, c \in [-1000; 1000]$, and returns 1 if the triangle is scalene, 2 if it is isosceles, 3 if it is equilateral, and 4 if it is not a triangle at all. Here, the arguments are uniformly distributed across the given range [41]. The probabilistic symbolic execution allows one to get a valuable insight about the behavior of the given code. For instance, what is the probability that the inputs of the function form a scalene or isosceles triangles?

Figure 2.10 presents the execution tree for the triangle problem. We first need to get a set of inputs for the triangle. Therefore, there are 1000 triangles with three equal sides: (1; 1; 1), (2; 2; 2), ..., (1000; 1000; 1000).

The probability that the function returns a scalene triangle or the assignment in line 9 is executed, will be 6.2125×10^{-2} . The probability for getting a set of inputs that forms isosceles and equilateral triangles will be 2.8045×10^{-4} and 1.2481×10^{-7} respectively, as described in Table 2.1.

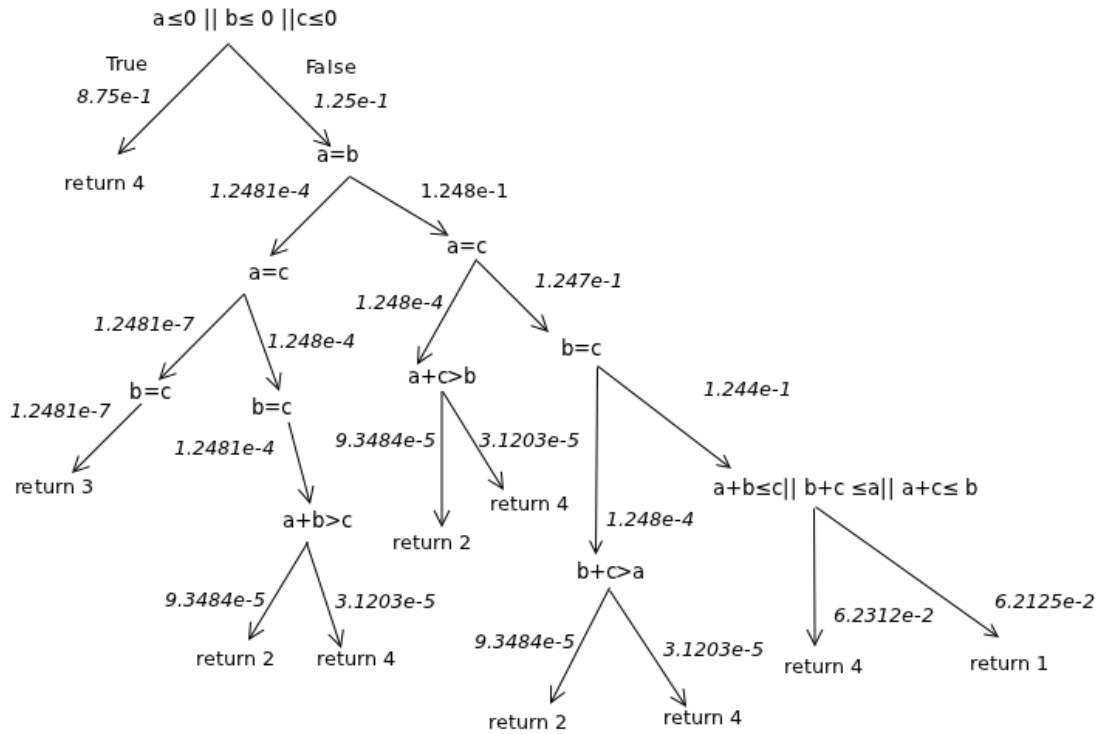


Figure 2.10: Probabilistic symbolic execution tree for the triangle problem.

Classifications	Probabilities
Scalene	6.2125×10^{-2}
Isosceles	2.8045×10^{-4}
Equilateral	1.2481×10^{-7}
Not triangle	9.3759×10^{-1}

Table 2.1: Classification and probabilities for the triangle problem [41].

2.5 Related Work

The work that is most closely related to this work is a technique recently presented by Geldenhuys et al. [41]. Others have taken a formal approach on procedural probabilistic reasoning [71] which we will not elaborate in this study. There are different number of works on static analysis and test case generation related to ours which we are discussing in this section.

2.5.1 Probabilistic Symbolic Execution

Geldenhuys et al. [41] proposed an approach that combines model counting and symbolic execution for performing probabilistic symbolic execution which enables the estimation of probabilities to program paths, and only supported the path conditions that could be expressed in Linear Integer Arithmetic (LIA) constraints.

Symbolic execution typically uses an interesting property that a path is either feasible/satisfiable or infeasible. However, The approach presented by Geldenhuys et al. [41] allows one to know how the path conditions which appeared to be satisfied can be considered with a certain probability i.e. add meaning to the values between 0 (infeasible) and 1 [41].

Doing probabilistic symbolic execution also required the inputs values to be uniformly distributed with their type, i.e., finite inputs domain. The model counting was used for counting the number of solutions to path conditions. The calculation of probabilities to path conditions include the counting of the number of these solutions and dividing them by the total space of values of the input domain size (the product of all the input domain size). The probabilistic symbolic execution (probsym), an extension to SPF allows one to calculate the path condition probability [41]. The formality of this approach forces one to think carefully about the effective model counting procedure and thus helps to count the solutions to a path condition yielding the probability for covering a certain portion of program.

While LattE has been used as model counter with the aim of supporting LIA. However, our work goes further to handle data structures and use Korat as a model counter for estimating the heap path conditions generated with lazy initialization algorithm.

The work of Geldenhuys et al. was also an efficient approach to present how effectively random testing works for a particular program. The aim was to present an extension of the used Symbolic PathFinder and to handle issues related to testing such as probability of obtaining coverage and discovering errors in programs but did not consider reliability as described in Section 2.5.2. Our approach builds on this work by also considering the generation of choices performed by lazy initialization and the counting of them.

In the approach presented, the probabilities were used to show how errors can be found by using the notion of the least likely paths through the code, how the chances of obtaining coverage can sometimes decrease and sometimes increase when input ranges are varied and lastly, how one can use the probabilities for fault localization [41].

2.5.2 Reliability Analysis in Symbolic PathFinder

In [39], the authors presented a technique to calculate the reliability of the software for supporting the analysis of structured data types, sequential and parallel programs. The reliability refers to the probability of the software to perform its assigned task requested by the user without having any failure [39]. The implemented analysis supports linear integer arithmetic operations, structured data types and concurrency. It was focused on extending SPF; this was performed such that the symbolic path finder extension cannot only detect errors (as it is currently) but also present the probability of encountering an error (or alternatively will give the probability that the program operates correctly). Their work was similar to this study. On one hand, it is limited to uniform distributions over finite data domains, LattE was used as model counting tool and the Korat algorithm was used as a model counting tool for structures. Also the effect of non-deterministic schedulers on multi-threaded programs was considered. On the other hand, this work goes beyond ours to use the concept of "confidence" as well as usage profile to estimate the probability of the paths conditions. To perform reliability analysis, two independent major tasks were accomplished :

1. Use SPF to generate path conditions and classify them in three categories. Those are: success, failure and grey conditions respectively.

2. Perform probabilistic analysis, that is performed through the use of a model counting tool. In this case, LattE and Korat tools were chosen to be used as model counters.

The above sets of path conditions indicated, form a complete portion of the entire domain [56]. Note that the input domains were considered to be finite and countable and also the success condition refers to the path condition that allow the execution of a program without occurrence of any error detected by SPF. The failure condition refers to the path conditions where there is occurrence of errors in the execution of a program. These errors can be run-time errors or deadlocks. The grey condition refers to the path condition for which the execution of a program is interrupted before its termination or detection of an error. The probability distribution were calculated based on the satisfiability of any of the successful path conditions. It has been reported that this technique can be applied to any symbolic execution approach e.g., KLEE [20], where the access to path conditions and thread schedules are possible [39]. The integration of usage profiles proposed by Filieri et al. to our method can allow us to move towards an approach that supports structures along with input probability distributions. Figure 2.11 shows the methodology used for the software reliability analysis in Symbolic PathFinder. Note that *Rel* refers to reliability.

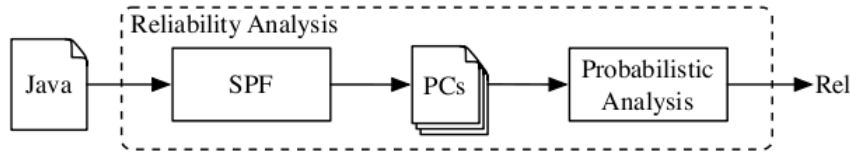


Figure 2.11: Reliability analysis methodology [39].

2.5.3 Program Analysis: From Qualitative Analysis to Quantitative Analysis

Liu et al. [61] proposed an approach that combines symbolic execution with volume computation for computing the exact execution frequency of program paths and branches. The volume computation was used to obtain the size of the solution space for the constraints. This technique points out the paths in a program that are executed more often than others. The proposed approach works well when the program paths that can be executed symbolically could lead to knowing how much input data that would drive the

program to be executed along a given path. Some of the quantitative program analysis methods based on volume computation and model counting are hot path detection, branch prediction and test case selection. With their approach, the path condition slicing and memoization were not developed, although mentioned. Wei et al. [95] also introduced a local search based on a method that uses Markov Chain Monte Carlo sampling to compute an approximation of the true model count from a given formula. The approx-count [44] model counter exploits the fact that if one can sample uniformly from the set of solutions of a formula F , then one can estimate the number of solutions. Unfortunately, there was no guarantee on the uniformity of the samples from the samplesat [44] model counter.

2.5.4 The Road not taken: Estimating Path Execution Frequency Staticly

Work done by Buse [19] proposed a method for estimating path execution frequency based on giving a statistical model which was based on syntactic features of the program's source code which has similar approach to this study. With semantic information in the program path the hot path was able to be identified, by combining symbolic execution with efficient constraints solving techniques. Volume computation was applied for accurate analysis, finding errors and generating test data. Besides this, the counting version of satisfiability modulo theories were presented [65] based on computing the volume of solution space on a set of boolean combinations of linear constraints and presenting how often a given program path is executed. De Loera et al. [65] generalizes model counting and volume computation problems for convex polytopes which have potential applications related to program analysis and verification [19].

2.5.5 Volume Computation for Boolean Combination of Linear Arithmetic Constraints

Ma et al. [65] generalized the problem of model counting and volume computation for convex polytopes in studying the counting version of satisfiability modulo theories, that is, how to compute the volume computation of solution space given a set of boolean combination of linear constraints. The difference between the method proposed by Buse in [19] is that the authors have used semantic information in the program paths which can calculate the exact probability of executing a path. The polytopes are referred to the bound intersection of finitely many halfspaces/inequalities and are normally described

using the H-representation. The halfspace (H) representation is concisely encoded as the matrix inequality: $x \mid AX \leq b$ (where A is a matrix of dimension $m \times d$ and b a vector of dimension m) [65]. H-representation is a natural representation for the conjunctive fragment of LIA except that it is not possible to directly express disequality constraints, e.g., $x \neq 0$ [41]. They described a method of analysing programs by checking the program's properties and processing individual paths in the flow graph of the programs. Their approach was based on computing the path condition for a program with a boolean formula. Here a model was considered as an assignment of truth values to all the boolean variables that led it to be evaluated as true, decide whether it is satisfiable or not, and compute the volume for the given formula. This has potential applications to program analysis and verification. The tool implemented allows for the computation of how often a given program path is executed but the focus was only on one application of volume computation technique.

2.5.6 Counting the Solutions of Presburger Equations without Enumerating them

Boigelot and Latour [17], addressed the problem of counting the number of distinct elements in a set of numbers or of vectors. They proposed an algorithm that enabled to produce an exact count without enumerating explicitly the vectors. The counting technique was based on constructing a number of decision diagrams that is considered as a finite state machine recognizing the encoding of integer vectors belonging to the set represented by Presburger arithmetic. The presburger arithmetic was used as a powerful formalism for solving the integer variables. Their approach handles the problematic projection operation and the result of construction procedure has been implemented and applied to problems involving a large number of variables. The problem of counting the number of solutions of a Presburger equation has been solved by Pugh [82] using a formula-based approach. More precisely, that solution proceeds by decomposing the original formula into a union of disjoint convex sums, each of them being a conjunction of linear inequalities. All variables, except one, are projected out successively by splintering the sums in such a way that the eliminated variables have one single and one upper bound. This eventually yields a finite union of simple formulas, on which the counting can be carried out by simple rules. Model counting is also extremely important in non-Boolean domains, including integer linear programming [13] and linear integer arithmetic, more details can be found in [17, 82].

2.5.7 Counting Models in Integer Domains

Morgado et al. [70] described the problem of counting models in Integer Linear Programming (ILP) using boolean satisfiability techniques. In their work, the proposed approaches were first based on encoding instance of Integer Linear Programming into Pseudo-Boolean (PB) instances, and alternatively based on encoding instances of ILP into instance of Satisfiability (SAT). This is mapping the Integer Linear Programming instances into Pseudo-Boolean constraints and then encoding the pseudo-boolean constraints into Satisfiability.

With the first approach, the lower and upper bounds on the possible values of integer variables were determined because the ILP instances define a convex polytopes, and every integer variable is guaranteed to have a lower and upper bound for performing model counting in pseudo-boolean formulation. With this method, all possible variable assignments were enumerated by using a backtrack search PB solver.

The second approach was proposed and based on encoding all pseudo boolean constraints into propositional clauses and use a SAT solver directly. This was done with the objective of taking an advantage of the powerful techniques of SAT solvers in dealing with the propositional clauses since they enable one to know the counting safety such that this can be used for model counting. Counting safety appears with the number of models in the PB formulation and in the case where the encoded SAT formulation are the same. As result, some Pseudo-Boolean (PB) to SAT encoding may overestimate the number of models, whereas others are shown to yield the correct number of models. Thus, the counting models in integer domains can be achieved by encoding ILP constraints into SAT directly using SAT solvers. Therefore, the PB counter is competitive with the SAT solvers.

2.6 Concluding Remarks

In this chapter we have provided information related to the symbolic execution for both integers and programs with heap objects. We also presented a general background of the Java PathFinder, an extensible framework for verifying Java bytecode. Symbolic PathFinder is a tool that uses symbolic execution, constraints solving as well as model checking to handle inputs of both reference and primitive types. Besides this, it performs the analysis of Java programs with unspecified inputs, up to a chosen search depth bounding (i.e., limiting the number of execution steps) [81]. The descriptions of model

counting for integers and structures are presented with the aim of establishing a standard approach for extending probabilistic symbolic execution.

The concepts of probabilistic symbolic execution which utilize model counting and symbolic execution techniques to calculate the execution of a program were presented with an example that illustrated them.

The chapter ended with a discussion on several related works, and comparison of the methods used in these works. In the next chapter, the methods used to extend the probabilistic symbolic execution (Section 2.4) are presented.

Chapter 3

Approach

Probabilistic symbolic execution (PSE) as described in Section 2.4 allows one to calculate path probabilities for programs manipulating integers. We shall here extend it to handle data structures, i.e. reference types. Note we make the same simplifying assumption as in Section 2.4 and in Geldenhuys et al. [41], namely, that inputs are uniformly distributed in the input domain, even though Filieri et al. [39] (summarized in Section 2.5.2) showed how this restriction can be relaxed.

There are two general approaches we consider to handle program manipulating complex data structures. Firstly, we assume data structures are in fact concrete and the only symbolic data is the data entered into the structures. This is essentially just the classic PSE described in Section 2.4. Secondly, we however consider that the actual input for the symbolic analysis also contains a symbolic structure. The contribution of this work lies in this second general approach.

When the structure is symbolic, we consider two approaches to calculating path probabilities for structures. The first approach simply makes all lazy initialization choices (see Figure 2.3) have the same probability throughout the execution. This approach is an approximation and is described in Section 3.3.1. The second approach we consider is precise and involves using Korat to precisely count the number of solutions to a structural constraint ("Heap Path Condition" in the terminology from Section 2.1.2) and a data constraint (Path Condition). This approach is described in Section 3.3.2. Figure 3.1 gives a graphical view of the approach.

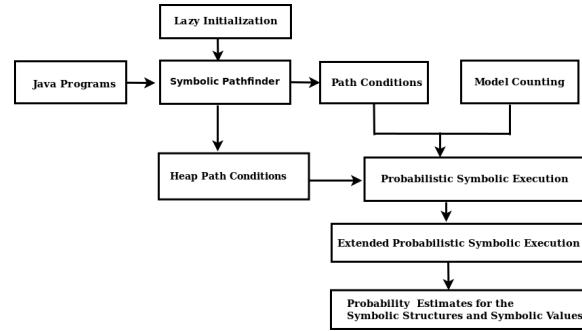


Figure 3.1: Extended probabilistic symbolic execution chain.

3.1 Symbolic Execution Drivers

Lets us first consider how the symbolic execution within Symbolic PathFinder (SPF) will be handled by looking at the drivers for the two general approaches mentioned above (symbolic values as input versus symbolic structures as input). Although we mention a generic container here as the data structure under analysis, we actually only focus on a Binary Search Tree in this work.

```

1 void runTest (int [] options, int limit) {
2   Container c = new container ();
3   int round = 0;
4   while (round < limit) {
5     if (options[round] == 0)
6       c.add(options[limit + round]);
7     else if (options[round] == 1)
8       c.find(options[limit + round]);
9     else
10      c.delete(options[limit + round]);
11    round++;
12  }
13 }
14
15 void runTestDriver (int length) {
16   int [] values = new int [length*2];
17   int i=0;
18   while (i < 2*length) {
19     if (i < length)
20       values[i] = Debug.makeSymbolicInteger("c" + i); //0,1,2
21     else
22       values[i] = Debug.makeSymbolicInteger("v" + i); //0..9
23     i++;
24   }
25   runTest(values, length);
26 }

```

Listing 3.1: Test driver used to run the code with symbolic values [41].

Figure 3.1 shows the code for the case where we analyze the system with symbolic values as input and where the actual container is kept concrete. The approach involves creating a number of symbolic variables dependent on the length of the sequence of actions you want to analyze. For instance, in the next chapter, the length of sequence of actions used to analyse the container is 4. The call to *Debug.makeSymbolicInteger* is intercepted by SPF to allow a symbolic variable of integer type to be created. The actions we consider here are add, delete and find. This driver will symbolically analyze all sequences of these actions (with symbolic inputs) up to the value of length. Internally all configurations of the containers will be created.

In Figure 3.2 we see the driver when we consider both the structure and the input value to be symbolic.

```

1 public static void runTestDriver(int size) {
2     Container b = new Container();
3     b = (Container)Debug.makeSymbolicRef("root",b);
4     if (b != null) {
5         b.add(Debug.makeSymbolicInteger("x"), size);
6         //b.find(Debug.makeSymbolicInteger("x"), size);
7         //b.delete(Debug.makeSymbolicInteger("x"), size);
8     }
9 }

```

Listing 3.2: Test driver used to run the code with a symbolic structure.

The symbolic structure that we consider here is the container itself and it is made symbolic with the *Debug.makeSymbolicRef* call (also intercepted by SPF). Since the symbolic structure will be initialized to *null* or a new concrete structure according to the lazy initialization rules, one must explicitly state that the call to the action must only happen when reference is not null, otherwise a null pointer exception will be thrown. Note also that since the structure is symbolic one is not required to call it multiple times, once is enough to capture all configurations of the structure up to a certain size. The size is given as a parameter to the calls, to ensure that only structures up to that size is considered. Lastly, we need to analyze the three actions one at a time, hence the lines commented out.

3.2 Classic Probabilistic Symbolic Execution

Here we use the classic PSE approach and assume that the structures we are analyzing are concrete, only the data is symbolic. Therefore we use the driver in Figure 3.1. The assumption is here that we drive the system in the way that it will be used in the real

world and, therefore, we get what one might term precise probabilities for each path's execution. The drawback of this approach is that we need a sequence of actions and thus it is not a modular solution.

3.3 Extended Probabilistic Symbolic Execution

Extended probabilistic symbolic execution is the contribution of the thesis. This technique considers the case where the input structure is taken to be symbolic, and not just the input data. Therefore, now we use the driver in Figure 3.2. We consider 2 approaches, first an approximate approach where we assign conditional probabilities across the execution and then perform a precise analysis where we calculate the probabilities using the Korat tool.

3.3.1 Fixed Choices

We consider the choices being made during the lazy initialization process and specify a formula for calculating the conditional probabilities in each case. Remember from Figure 2.3 that there are essentially 3 choices to make whenever a symbolic reference field is being accessed during lazy initialization.

New object : where a new object of the desired type is created with symbolic fields

Null: where the reference is assigned the *null* value

Alias options: where the reference is nondeterministically assigned each object of the desired type that has previously been generated during the analysis.

In SPF these choices are captured within a `HeapChoiceGenerator` and it will know precisely how many options exists, since the number of possible alias options are tracked. There will be a minimum of two options (null and new) if it is the first time the `HeapChoiceGenerator` is exercised. If the first visit resulted in a new object being created then any subsequent use of the `HeapChoiceGenerator` will have at least 3 options: null, new and a number of alias options. We, therefore, assign probabilities to these options by only assigning a probability for null (called *nullP*) and new (called *newP*) and deriving the probabilities for the choices as follows:

Initial 2-option case: Assign the probability for taking the null option as *nullP* and the case for the new object as $1 - \text{nullP}$.

3+ option case: Assign the probability for taking the null option as $nullP$, the new option as $newP$ and the alias options are uniformly distributed as $\frac{1-nullP-newP}{total-2}$. Note that $total$ is the number of choices in total for the HeapChoiceGenerator (for example if there are 3 alias possibilities then $total = 5$).

By assigning the probabilities in this way, whenever a lazy initialization is performed the precise conditional probability can be calculated. This is a very simple approach to introduce probabilities in the context of symbolic structures, but of course it is not very precise. For example, it should be clear that having a reference as null is context dependent and is unlikely to always have the same probability. In the case of a linked list, it is more likely that a reference close to the end of a list will be null, than one at the first node. Next, we consider a more precise approach.

3.3.1.1 Example

Listing 3.3 shows a fragment of code to illustrate the idea behind the probabilistic symbolic execution for a program with structures as inputs. Figure 3.2 presents its execution probabilities tree where we assigned the probabilities to different nondeterministic choices ($null$, new , $alias$) generated during the lazy initialization process. The ($null$, new , $alias$) distributions used are (10, 80, 10) (i.e. 10% null, 80% new and the rest, also 10%, as alias options). Furthermore we only consider two data values (0, 1) for the integer *elem* field.

```

1 class Node {
2     int elem;
3     Node next;
4
5     Node swapNode() {
6         if(next!=null) {
7             if(elem > next.elem) {
8                 Node t = next;
9                 next = t.next;
10                t.next = this;
11                return t;
12            }
13        }
14        return this;
15    }
16
17    public static void main (String[] args) {
18        Node n = new Node();
19        n = (Node) Debug.makeSymbolicRef("input_n", n);
20        if (n != null)
21            n = n.swapNode();
22    }

```

Listing 3.3: Source code for swapping a node.

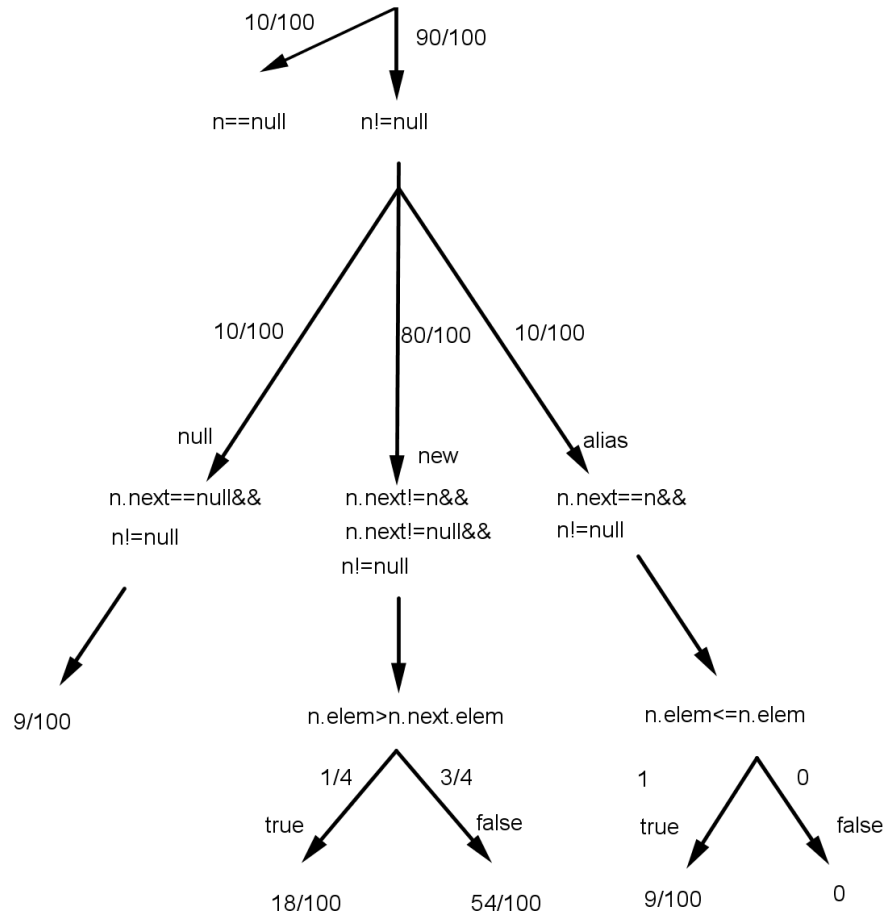


Figure 3.2: Probabilistic symbolic execution tree for the code in Listing 3.3.

Figure 3.2 shows the conditional (on the branches) and path (end of each path) probabilities of the structural and data choices for the symbolic execution of the code in Listing 3.3. Note that we create a symbolic *Node* with the *Debug.makeSymbolicRef* call and then call the *swapNode* method on this symbolic instance. The top choice in the tree comes from the check on line 20 to see if the instance is *null*. Since there are only 2 choices here we assign the non-null option $1 - 0.1$, expressed as 90/100 in Figure 3.2. The next point where there is lazy initialization is when the *next* field is first accessed on line 6 in *swapNode*; this is shown in the 3-way split in the middle of the tree. There are now 3 choices since an alias option is also possible (and shown on the right-hand branch). The probabilities now directly follow the initial null, new and alias values of 10%, 80% and 10% respectively. For the left-most branch where the *next* field is null, the code terminates. For the other two branches, the data constraint from line 7 must now be evaluated. Lets first consider the alias case (right-most branch): clearly *n.elem*

$> n.next.elem$ is infeasible when $n == n.next$. For the middle branch, where $n.next$ is a new symbolic object, there are 4 options induced by the data constraint at line 7, but only one of them (where $n.elem = 1$ and $n.next.elem = 0$) leads to line 8. Hence the probability to reach line 8 (Listing 3.3) in the code, when we have only 2 data values is 0.18.

3.3.2 Model Counting with Korat

In Section 2.3.2 the Korat tool was introduced for counting structures. It requires two inputs: bounds on the size of the structure (called the finitization) and a predicate describing legal structures (called the representation invariant, or *repOk* for short). Taking these two inputs Korat will enumerate all legal structures within the bounds. We are interested only in how many legal structures there are.

We assume that the size of structures we are interested in is fixed for the duration of the analysis, hence we encode this once off in the finitization. For example, Listing 2.6 shows the finitization for a Binary Tree where the number of nodes allowed are fixed to be *NUM_Node*.

Listing 2.6 also shows a *repOk()* function for a Binary Tree. Recall that for integers we needed to know exactly how many values there are in the compete domain of all variables involved in the path condition to calculate the probability of the path condition being satisfied. Here it works similarly, we need to know how many structures are possible if there are no constraints. This can be calculated by replacing the actual *repOk()* by one that simply returns *true* whenever it is called. In essence this only uses the finitization bounds to calculate the number of structures.

When calculating the probability of a path condition during the execution of SPF, we must now consider both the path condition (that encodes constraints on the integer variables) and the heap path condition (that encodes constraints on the structures). The former we refer to simply as PC and the latter as HeapPC. Note that the PC can encode constraints on the integer variables stored within the structures. This dependency is unfortunate, since without it, one could calculate the probabilities of the PC and HeapPC separately and simply multiply them. Here, however, we can now have a PC that refers to constraints that are trivially false for certain structures. An example would be where aliasing is allowed and a constraint states $node.value > node.next.value$; any structure where $node.next = node$ would not be valid due to the constraint being false (more importantly for us the solution count would be zero). This means we now need a single

procedure to count solutions, and, since we have structures to count we need an approach such as Korat to do it.

Naturally, if we have integer constraints that are independent with respect to the symbolic structures, then we can still use the more efficient LattE-based counting, i.e. only constraints that are dependent on the structure need to be counted by Korat, but when that happens Korat must also enumerate the integer domains within the structures. Unlike LattE, Korat's runtime complexity is dependent on the size of the integer domains [18].

We transform the typical *repOk* used by Korat to incorporate the PC and HeapPC as seen in Listing 3.4. *PC'* here refers to the part of the path condition that is dependent (including transitively) on the structures and *repOKOriginal()* encodes the actual structural constraints (for example those in the *repOK* from Listing 2.6 for a Binary Tree).

```

1 public boolean repOK() {
2     return HeapPC && PC' && repOKOriginal();
3 }

```

Listing 3.4: Predicate method (*repOK*) for counting.

In a slight abuse of notation (by giving *repOk* an input constraint) we can now calculate the probability of a PC and HeapPC is calculated in the equation below. Korat refers to calling the tool with a *repOK* that either calculates the complete domain size (i.e. the *true* case) or the case described in Listing 3.4, similarly, *LattE* refers to calling the tool with *PC''* that refers to the part of PC that doesn't contain any transitive dependencies on anything in the structure encoded by HeapPC ($PC = PC' \wedge PC''$). *Domain* refers to the product of the sizes of the domains of the variables in *PC''*.

$$probabilityKorat(PC \wedge HeapPC) = \frac{Korat(repOK(HeapPC \wedge PC'))}{Korat(repOK(true))} \times \frac{LattE(PC'')}{Domain} \quad (3.3.1)$$

3.3.2.1 Example

Consider the source code for swapping a node shown in Listing 3.3, we now present an example that illustrates the use of Korat to calculate the conditional and path probabilities. Korat requires a finite domain, hence we again assume data values are either 0 or 1 and here we need to add one additional constraint on the size of the linked list which we assume will have only 2 nodes. These constraints are passed to Korat as its finitization function. Listing 3.5 shows an example of a *repOk* used during the analysis and this one

specifically enumerates the structures to reach line 8 3.3 in the code. The total number of valid structures for the linked list under the given finitization is 17 (i.e. when *repOk* is simply *true*).

```

1 public boolean repOK() {
2   return n!=null && n.next!= null && n.next!=n && n.elem>n.next.elem
3 }
    
```

Listing 3.5: Predicate method (*repOk*) to reach line 8 in Listing 3.3.

Figure 3.3 shows the probabilistic symbolic execution tree for the code shown in Listing 3.3. The probabilities are calculated using the Korat tool. This figure is the counterpart of Figure 3.2 that was created based on fixed probabilities for each lazy initialization choice. Note that all the probabilities shown are path probabilities and since there are 17 total structures we express each probability as how many of the 17 structures lead to that choice. Korat counts both the data and structural constraints together, but for illustration purposes, we split them giving the structures first and then a data choice if it exists, but only one probability incorporating both results. Lastly, to illustrate how Korat works on this example we also show the actual structures that Korat counts for each case.

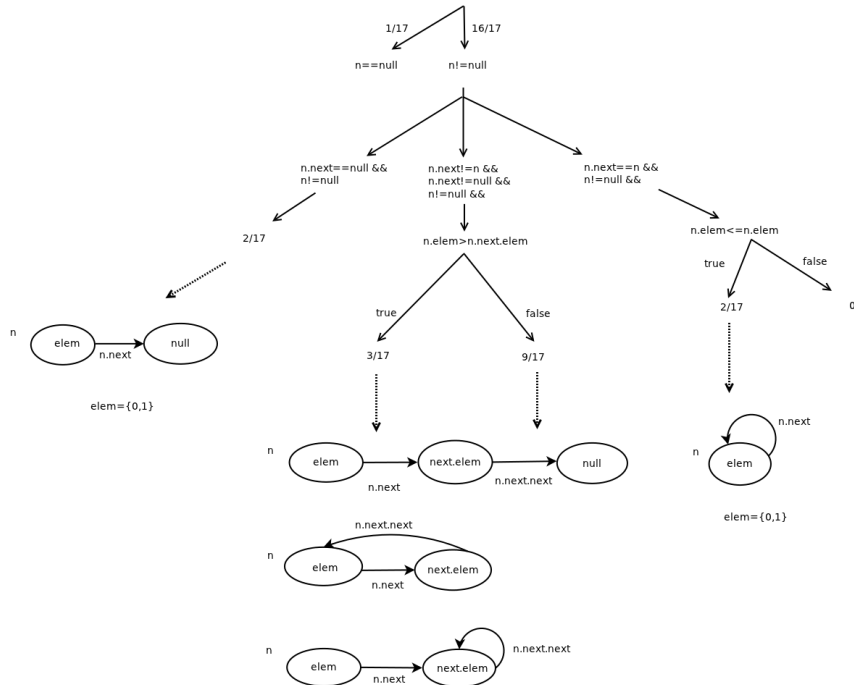


Figure 3.3: Probabilistic symbolic execution tree for the code in Listing 3.3 using Korat.

As before, the top split in the tree represents the null check outside of *swapNode* and 16 out of the 17 structures pass the point and the interesting options occur from the lazy

initialization of the *next* field. On the left we see the case where the field is null and since *elem* field can be either 0 or 1 for this structure, Korat returns 2 structures and the path probability for this case is $2/17$. Recall that in the fixed choices case this probability was $1/10$. The right-most branch is again the alias case and as with the null case there are two data value options and thus again it has a $2/17$ probability. The middle case is the interesting one and the *repOk* passed to Korat in the case where $n.elem > n.next.elem$ is assumed true is shown in Listing 3.5. The result in this case is 3, since there is only one data assignment that satisfies each structure (where $n.elem = 1$ and $n.next.elem = 0$). However on the other branch there are 3 solutions for each structure and thus 9 valid structures returned by Korat.

Notice how closely the results from Figure 3.2 where we used the fixed choices of (10, 80, 10) matches¹ the results from Figure 3.3 which came from the precise analysis based on Korat. As results in the next chapter will also show, it is often the case that low probabilities for null and alias options compared to the new option matches real-world behavior.

3.4 Implementation

The approach where we use fixed probabilities (Section 3.3.1) is implemented in the open-source repository on Google Code at probsym.googlecode.com. It is a simple extension of the classic approach (Section 3.2).

The implementation is based on a Listener that is triggered whenever a path probability is to be calculated. It exploits the fact that the probabilities of each choice generator is independent (in this case, but not in the Korat approach) and thus we can independently calculate the probability for each ChoiceGenerator and simply multiply them. It also uses the fact that in JPF ChoiceGenerators are chained together so it is simple to traverse all of them up to the point in the path where we want to calculate the path probability. A code skeleton is shown in Listing 3.6. Although not shown, there is caching within the *calcProbabilityLattE* call to ensure it is optimal.

¹ $2/17$ is approximately 11.7% and $3/17$ is approximately 17.6%

```

1 private Apfloat extendedProbCalc(ChoiceGenerator<?> cg) {
2     Apfloat pr = new Apfloat(1.0,PRECISION);
3     Apfloat p;
4     while (cg != null) {
5         if (cg instanceof PCChoiceGenerator) {
6             PathCondition pc = ((PCChoiceGenerator) cg).getCurrentPC();
7             p = calcProbabilityLattE(pc);
8         }
9         else if (cg instanceof HeapChoiceGenerator) {
10             HeapChoiceGenerator hcg = (HeapChoiceGenerator)cg;
11             int choices = hcg.getTotalNumberOfChoices();
12             int currentChoice = hcg.getNextChoice();
13             if (choices == 2) { // only null and new, then use newProb=1-nullProb
14                 if (currentChoice == 0) { //null
15                     p = nullProb;
16                 } else { // currentChoice == 1, i.e. new
17                     p = Apfloat.ONE.subtract(nullProb);
18                 }
19             } else if (choices > 2) {
20                 if (currentChoice == 0) { // null
21                     p = nullProb;
22                 } else if (currentChoice == 1){ //new
23                     p = newProb;
24                 } else { // 2 or more, alias
25                     Apfloat aliasProb = Apfloat.ONE.subtract(nullProb).subtract(
26                         newProb);
27                     p = aliasProb.divide(new Apfloat(choices - 2,PRECISION));
28                 }
29             } else {
30                 p = Apfloat.ONE.divide(new Apfloat(choices, PRECISION));
31             }
32         }
33         else { // all other forms of choices are just uniform
34             int choices = cg.getTotalNumberOfChoices();
35             p = Apfloat.ONE.divide(new Apfloat(choices, PRECISION));
36         }
37         pr = pr.multiply(p);
38         cg = cg.getPreviousChoiceGenerator();
39     }
40     return pr;
41 }

```

Listing 3.6: Code Skeleton for Fixed Choices.

An implementation of the Korat approach to counting (Section 3.3.2) is complicated by the fact that the choices are now no longer independent. Now, one first needs to extract the complete PC and HeapPC from the current state and then make the *calcProbabilityLattE* call. One also needs to generate the code that needs to go into the *repOK* method by analyzing the PC and HeapPC, then compile the code and use as input to Korat. This

is currently not completely implemented and for the results in the next chapter, this step is performed by hand.

3.5 Concluding Remarks

We believe an approach that enumerates all valid structures, is one key way to accurately calculate probabilities for symbolic structures. The approach where we fix the conditional probabilities can be seen as a fast approximation of the probability. Korat, which has a long history in software engineering and testing literature, achieves our stated goal of counting structures without having to change the tool in any way. The drawback of using Korat, however, is that it doesn't handle integer domains very efficiently, since it enumerates them. The ideal approach would be to combine a Korat-like approach that enumerates structures with a more efficient approach to calculating solutions to the integer portions of the constraints (such as LattE for example). One approach to create such a combination is to enumerate each structure and then to use LattE to count the valid integer solutions for each structure. However using Korat in this fashion is not easy and one should rather build a new tool to do this. We intend to use the approach to doing bounded lazy initialization [42] within SPF to build a new tool to enumerate structures efficiently. This will require a call to SPF from within the probabilistic symbolic execution listener which in turn is listening to SPF itself.

Chapter 4

Results and Discussions

In this chapter we present the results obtained from conducting experiments using the 3 techniques described in the previous chapter. We use an implementation of a Binary Search Tree adapted from [91] (see Appendix A).

We used a specific application of probabilistic symbolic execution, also used in [41], that calculates the probability of reaching specific points in the code. More specifically it collects a set of all the paths that reach the location of interest and then calculates the sum of these probabilities, as well as the maximum probability and minimum probability from the set. These calculations are conducted within a Listener for SPF, which itself forms part of an extension called *JPF-Probsym*. We focus on the *add*, *delete* and *find* methods in the container.

We used results from classic probabilistic symbolic execution (referred to as symbolic values) as our baseline in all the results. Although this is not directly comparable, since it measures probabilities during actual use of the container, rather than the modular approach we focus on in this thesis where input structures are also symbolic. The results show that the trends one observes when using symbolic values mostly aligns with the symbolic structures approach.

We run each test with varying sizes for the data domains for the variables to add to the container. For the symbolic values we look at a sequence length of 4 actions, and for the symbolic structures we consider only 3 nodes. Although this might seem like a discrepancy, it is actually the case that 4 actions match 3 nodes quite well. For example when we analyze a *delete* operation with 3 nodes in the symbolic structure case, one would actually require the 4 actions *add;add;add;delete* to observe similar behavior in the symbolic value case.

All experiments were carried out on a machine with the following hardware and software configurations:

- Core 2 Duo configuration: Intel Core 2 Dual CPU, 3.06 GHz and 2.9 GB of RAM. It runs the Ubuntu 11.10 operating system, 32-bit running on Linux 3.0.0.
- JPF model checker version 6.0 with open Java Development Kit Version 1.7.0.
- SPF and JPF-Probsym.

4.1 Fixed Probabilities

In the previous chapter we saw that picking fixed probabilities can be relatively accurate, but that was for a very simple example. Here we consider a more realistic scenario and we run the analysis with varying probabilities for $(null, new, alias)$. We report in Table 4.1 the results when comparing fixed probabilities against the baseline of using classic probabilistic symbolic execution for sequences of 4 actions and data values in the range $[0..9]$. Instead of showing all combinations of fixed probabilities we only show the ones with the “closest” match to the classic case (per method). We calculate the best match by looking at the average distance from the expected value (classic) for each combination of fixed probabilities per method. Note that we use the sum of the path probabilities to reach each location; using either the minimum or maximum probability does not work here since they are equal to the expected value in almost all cases.

Method	Location	Symbolic Values		Fixed Probabilities		
		Paths	Probability	Paths	Probability	(Null,New,Alias)
Add	1	65	4.4549×10^{-1}	7	2.0026×10^{-1}	(10,89,1)
	2	81	5.5893×10^{-1}	33	2.7403×10^{-1}	
	3	65	4.4549×10^{-1}	7	2.0026×10^{-1}	
	4	81	5.5893×10^{-1}	33	2.7403×10^{-1}	
Find	5	81	1.4024×10^{-1}	7	5.9809×10^{-2}	(10,70,20)
	6	81	5.5893×10^{-1}	13	3.4654×10^{-1}	
	7	81	5.5893×10^{-1}	13	3.4654×10^{-1}	
Delete	8	117	5.5893×10^{-1}	13	2.6747×10^{-1}	(5,90,5)
	9	117	5.5893×10^{-1}	13	2.6747×10^{-1}	
	10	130	8.9099×10^{-1}	14	4.0551×10^{-1}	
	11	23	7.0499×10^{-2}	1	4.0500×10^{-2}	
	12	14	1.7062×10^{-2}	11	7.1074×10^{-7}	
	13	14	1.7062×10^{-2}	11	7.1074×10^{-7}	
	14	12	1.5562×10^{-2}	7	1.2437×10^{-2}	
	15	1	7.4999×10^{-4}	60	2.0321×10^{-4}	
	16	1	7.4999×10^{-4}	54	9.6159×10^{-5}	
	17	12	1.5562×10^{-2}	7	1.4625×10^{-2}	
	18	1	7.4999×10^{-4}	54	9.6396×10^{-5}	
	19	1	7.4999×10^{-4}	60	2.0344×10^{-4}	

Table 4.1: Probability of covering locations in Binary Tree [0..9]

A discussion of results in Table 4.1 follows.

The first column specifies the methods we are using and the second column refers to the locations being reached. The third column (Symbolic Values) shows the number of paths explored for each of the locations being reached as well as the probability for that happening using the classic probabilistic symbolic execution. These probabilities are considered the baseline and represents the typical use-case for the container. The fourth column (Fixed Probabilities) shows the number of paths explored to reach the locations and the probability for that happening using fixed probabilities for every lazy initialization option. Lastly, it also contains the parameters that we fixed for (null, new alias) choices presented in coordinate form, (x, y, z) . These parameters contain the values of which the results obtained (using symbolic structures) are closest to our target results (those obtained using symbolic values).

The classic approach in the symbolic values column is quite fast when only sequences of 4 actions are considered (less than 10 mins). For the fixed probabilities one analyzes each method individually, which of course scales much better (each method by itself took less than 2 mins). For the remainder of the results in this chapter we don't report running time, since the techniques for Korat are not fully automated.

The first observation to make is that as with the example in the previous chapter, the best matching results are obtained with null probability low (10% or less), new probability high and alias probability again quite low. One can also see that although the probability for reaching locations are different they match on which locations have the same probability. The exceptions to this are noteworthy, since one will intuitively expect them to always be the same, in fact in the results in the rest of this chapter where we consider minimum and maximum probabilities for locations, they are the same. This indicates the drawback of using these fixed probabilities: they can be inaccurate! This is largely due to the fact that some structures were not realized. For example one would expect location 15 and 16, as well as 18 and 19 to be the same but they are not (in the rest of the chapter these are the same).

Note how there is no known connection between the number of paths that reach a location and the probability for reaching a location. This is an interesting feature of probabilistic symbolic execution in general and it shows how much we can learn in terms of program understanding from these techniques. We know that the fixed probabilities is just an approximation of the real probabilities, so we refrain from making any other observations on them and rather study next the precise probabilities calculated with Korat.

4.2 Korat Approach

In this section, we present the maximum and minimum probabilities for reaching the locations in the binary tree example with the use of the Korat approach (symbolic structures) and compare it to the classic approach (symbolic values). For the symbolic values we used a sequence length of 4 actions and we bound the structures for Korat to 3 nodes. We vary the range of data values between $[0..9]$, $[0..14]$, $[0..49]$ and $[0..99]$. We didn't consider any aliasing in the structures (encoded in *repOkOriginal* in Listing 3.4). The maximum probabilities for reaching locations are presented in Tables 4.2 and 4.3.

Method	Location	Paths	0...9	0...14	0...49	0...99
Add	1	65	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
	2	81	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
	3	65	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
	4	81	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
Find	5	81	2.5000×10^{-2}	1.6666×10^{-2}	5.0000×10^{-3}	2.5000×10^{-3}
	6	81	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
	7	81	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
Delete	8	117	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
	9	117	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
	10	130	1.1250×10^{-1}	1.1666×10^{-1}	1.2250×10^{-1}	1.2375×10^{-1}
	11	23	2.5000×10^{-2}	1.6666×10^{-2}	5.0000×10^{-3}	2.5000×10^{-3}
	12	14	5.6249×10^{-3}	3.8888×10^{-3}	1.2249×10^{-3}	6.1875×10^{-4}
	13	14	5.6249×10^{-3}	3.8888×10^{-3}	1.2249×10^{-3}	6.1875×10^{-4}
	14	12	5.6249×10^{-3}	3.8888×10^{-3}	1.2250×10^{-3}	6.1875×10^{-4}
	15	1	7.4999×10^{-4}	5.6172×10^{-4}	1.9599×10^{-4}	1.0106×10^{-4}
	16	1	7.4999×10^{-4}	5.6172×10^{-4}	1.9599×10^{-4}	1.0106×10^{-4}
	17	12	5.6249×10^{-3}	3.8888×10^{-3}	1.2250×10^{-3}	6.1875×10^{-4}
	18	1	7.4999×10^{-4}	5.6172×10^{-4}	1.9599×10^{-4}	1.0106×10^{-4}
	19	1	7.4999×10^{-4}	5.6172×10^{-4}	1.9599×10^{-4}	1.0106×10^{-4}

Table 4.2: Symbolic Values: Maximum probabilities for locations in Binary Tree

Method	Location	Paths	0...9	0...14	0...49	0...99
Add	1	7	6.1927×10^{-2}	6.4163×10^{-2}	6.7289×10^{-2}	6.7958×10^{-2}
	2	10	3.2614×10^{-1}	3.3834×10^{-1}	3.5542×10^{-1}	3.5908×10^{-1}
	3	7	6.1927×10^{-2}	6.4163×10^{-2}	6.7289×10^{-2}	6.7958×10^{-2}
	4	10	3.2614×10^{-1}	3.3834×10^{-1}	3.5542×10^{-1}	3.5908×10^{-1}
Find	5	7	9.9999×10^{-2}	6.6666×10^{-2}	1.9999×10^{-2}	0.9999×10^{-3}
	6	14	3.2614×10^{-1}	3.3834×10^{-1}	3.5542×10^{-1}	3.5908×10^{-1}
	7	14	3.2614×10^{-1}	3.3834×10^{-1}	3.5542×10^{-1}	3.5908×10^{-1}
Delete	8	7	4.4999×10^{-1}	4.6666×10^{-1}	4.8999×10^{-1}	4.9499×10^{-1}
	9	14	3.2614×10^{-1}	3.3834×10^{-1}	3.5542×10^{-1}	3.5908×10^{-1}
	10	14	6.1927×10^{-2}	6.4163×10^{-2}	6.7289×10^{-2}	6.7958×10^{-2}
	11	1	1.2187×10^{-6}	3.6177×10^{-7}	9.7931×10^{-9}	1.2248×10^{-9}
	12	11	8.9399×10^{-4}	6.1537×10^{-4}	1.9266×10^{-4}	9.1688×10^{-5}
	13	11	8.9399×10^{-4}	6.1537×10^{-4}	1.9266×10^{-4}	9.1688×10^{-5}
	14	1	1.3759×10^{-2}	9.1654×10^{-3}	2.7464×10^{-3}	1.3728×10^{-3}
	15	12	3.5098×10^{-3}	2.4310×10^{-3}	1.5048×10^{-3}	1.5685×10^{-3}
	16	11	3.5098×10^{-3}	2.4310×10^{-3}	7.6777×10^{-4}	3.8802×10^{-4}
	17	1	1.3759×10^{-2}	9.1654×10^{-3}	2.7464×10^{-3}	1.3728×10^{-3}
	18	11	3.5098×10^{-3}	2.4310×10^{-3}	7.6777×10^{-4}	3.8802×10^{-4}
	19	11	3.5098×10^{-3}	2.4310×10^{-3}	7.6777×10^{-4}	3.8802×10^{-4}

Table 4.3: Symbolic Structures: Maximum probabilities for locations in Binary Tree

A first interesting observation to make is a general one for both techniques: depending on the range of values allowed, some locations are easier to cover and some other locations more difficult. For example, adding values to the tree is easier when the range increases, since it is less likely to try and add an element already in the tree; in contrast, finding an element in a tree becomes more difficult since the likelihood of the exact value being in the tree is less if the range of values is larger. When comparing the results in Table 4.2 and Table 4.3 it is, however, clear that these trends are consistent for both approaches.

Another general observation that is evident in all the data, is the number of paths reaching a location doesn't imply anything about the probability of reaching the location. Sometimes a location is reached much more often than another and the probability for reaching either is the same (location 1 and 10 for example).

If we consider the code for the *Add* method in Appendix A we can see that location 1 and location 3 are symmetric and also 2 and 4; the symmetry is around whether the value to be added is less or greater than the value of the current node in the tree. Notice how this symmetry is clearly visible in the data for the symbolic structures (Table 4.3), but when considering the actual use of the container when adding symbolic values reaching all 4 locations are equally likely. This is a case where the over approximation of considering all possible structures gives us the intuitive result, but when we use the container, it seems not all these structures are realized.

According to the symbolic structure analysis, location 11 is the least likely to be reached, as can be seen in the code presented (see Appendix A), this is the case where we try and delete the root node, when both the left and right children are null. This clearly will be a rare case and also, as the data shows, will become more rare when more data values are allowed. Again, however, when considering the actual use of the container this location is not as hard to reach as some of the other ones.

Method	Location	Paths	0...9	0...14	0...49	0...99
Add	1	65	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
	2	81	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
	3	65	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
	4	81	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
Find	5	81	6.2500×10^{-5}	1.8518×10^{-5}	5.0000×10^{-7}	6.2500×10^{-8}
	6	81	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
	7	81	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
Delete	8	117	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
	9	117	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
	10	130	2.8125×10^{-4}	1.2962×10^{-4}	1.2250×10^{-5}	3.0937×10^{-6}
	11	23	6.2500×10^{-5}	1.8518×10^{-5}	5.0000×10^{-7}	6.2500×10^{-8}
	12	14	2.8124×10^{-4}	1.2962×10^{-4}	1.2249×10^{-5}	3.0937×10^{-6}
	13	14	2.8124×10^{-4}	1.2962×10^{-4}	1.2249×10^{-5}	3.0937×10^{-6}
	14	12	2.8124×10^{-4}	1.2962×10^{-4}	1.2249×10^{-5}	3.0937×10^{-6}
	15	1	7.4999×10^{-4}	5.6172×10^{-4}	1.9599×10^{-4}	1.0106×10^{-4}
	16	1	7.4999×10^{-4}	5.6172×10^{-4}	1.9599×10^{-4}	1.0106×10^{-4}
	17	12	2.8124×10^{-4}	1.2962×10^{-4}	1.2249×10^{-5}	3.0937×10^{-6}
	18	1	7.4999×10^{-4}	5.6172×10^{-4}	1.9599×10^{-4}	1.0106×10^{-4}
	19	1	7.4999×10^{-4}	5.6172×10^{-4}	1.9599×10^{-4}	1.0106×10^{-4}

Table 4.4: Symbolic Values: Minimum probabilities for locations in Binary Tree

Method	Location	Paths	0...9	0...14	0...49	0...99
Add	1	7	4.2188×10^{-3}	4.9162×10^{-3}	6.0193×10^{-3}	6.2743×10^{-3}
	2	10	4.2188×10^{-3}	4.9162×10^{-3}	6.0193×10^{-3}	6.2743×10^{-3}
	3	7	4.2188×10^{-3}	4.9162×10^{-3}	6.0193×10^{-3}	6.2743×10^{-3}
	4	10	1.6511×10^{-2}	1.8534×10^{-2}	2.1532×10^{-2}	2.2199×10^{-2}
find	5	7	3.7438×10^{-3}	2.8092×10^{-3}	9.8275×10^{-4}	2.5350×10^{-4}
	6	14	4.2188×10^{-3}	4.9162×10^{-3}	6.0193×10^{-3}	6.2743×10^{-3}
	7	14	4.2188×10^{-3}	4.9162×10^{-3}	6.0193×10^{-3}	6.2743×10^{-3}
Delete	8	7	1.6874×10^{-2}	1.9664×10^{-2}	2.4077×10^{-2}	2.5097×10^{-2}
	9	14	4.2118×10^{-3}	4.9162×10^{-3}	6.0193×10^{-3}	6.2743×10^{-3}
	10	14	4.2118×10^{-3}	4.9162×10^{-3}	6.0193×10^{-3}	6.2743×10^{-3}
	11	1	1.2178×10^{-6}	3.6177×10^{-7}	9.7931×10^{-9}	1.2248×10^{-9}
	12	11	2.3399×10^{-4}	1.7557×10^{-4}	6.1422×10^{-5}	3.1688×10^{-5}
	13	11	2.3399×10^{-4}	1.7557×10^{-4}	6.1422×10^{-5}	3.1688×10^{-5}
	14	1	1.3759×10^{-2}	9.1654×10^{-3}	2.7464×10^{-3}	1.3728×10^{-3}
	15	12	1.0529×10^{-3}	1.2290×10^{-3}	7.6777×10^{-4}	3.8802×10^{-4}
	16	11	3.5098×10^{-3}	2.4310×10^{-3}	7.6777×10^{-4}	3.8802×10^{-4}
	17	1	1.3759×10^{-2}	9.1654×10^{-3}	2.7464×10^{-3}	1.3728×10^{-3}
	18	11	3.5098×10^{-3}	2.4310×10^{-3}	7.6777×10^{-4}	3.8802×10^{-4}
	19	11	3.5098×10^{-3}	2.4310×10^{-3}	7.6777×10^{-4}	3.8802×10^{-4}

Table 4.5: Symbolic Structures: Minimum probabilities for locations in Binary Tree

Most of the observations made earlier about the maximum probabilities also hold true for the minimum probabilities shown in Table 4.4 and Table 4.5. Notice, however, that the probabilities are now lower, unless of course only one path reached the location in which case the maximum and minimum probabilities are the same.

If we consider the probabilities for locations 18 and 19, one will notice that the maximum and minimum probabilities are the same. This strange effect is due to the fact that although there are 11 paths reaching the respective locations the *repOk* is violated for 10 of those, i.e. only 1 path is considered feasible. This is a common occurrence during lazy initialization, since it is not aware of any complex structural constraints for the data structures.

4.3 Discussion

One can argue whether the results with the symbolic structures is in any way better than the ones where only the values are considered symbolic. This is however not our focus, since we only show how one can extend the classic approach to reason compositionally with the use of symbolic structures.

We give two solutions, one naive solution that allows only fixed probabilities for every choice, which we know is not as accurate as using Korat to calculate precise probabilities. However, the naive approach is very fast and scalable. Korat enumerates each possible structure, including all the data values. This means the larger the data domains, the longer Korat runs. When we used Korat, the smallest amount of time spent calculating the maximum probabilities (similar behavior was observed for the minimum case) was 2 seconds with the value ranges from $[0..9]$, 23 minutes were spent while getting the probabilities with the range of values $[0..49]$ and the longest time spent was 6 hours for range of values $[0..99]$. Latte on the other hand is insensitive to the size of data domains so the naive approach doesn't suffer from this scalability issue.

Chapter 5

Conclusions and Future Work

In this chapter, an overview of the research presented in this thesis, emphasizing the important achievements describing possibilities and some ideas for future work, are given.

The focus of the thesis has been on extending the existing probabilistic symbolic execution framework for handling data structures, thus, supporting reference types. Specifically, we added two new approaches for handling references, by building on the lazy initialization of Symbolic PathFinder (SPF). SPF is one of the only symbolic execution frameworks that handle data structures as symbolic input, and thus was the natural framework for us to build upon. The main contribution of the thesis is in Chapter 3 where we describe the two extensions.

The first approach was a very simple extension that adds fixed probabilities to each choice during lazy initialization (i.e. a fixed probability for extending a symbolic reference to be null, a new object or an alias to an existing object). The benefit of this approach is that it seamlessly integrates with the existing probabilistic symbolic execution approach in SPF. This in turn means it can use a very efficient approach to using model counting for linear integer constraints. The downside is that it is inaccurate, since in actual execution the probabilities for these choices are not fixed.

The second approach is considerably more complicated, but gives accurate results. It uses the Korat tool to count the number of structures that exactly satisfies the current path condition (including constraints on the reference types, the so-called *HeapPC*). The drawback of this approach is that Korat enumerates all structures during the counting, which means that if there are data fields, for example an integer, with a large domain, Korat slows down dramatically. In fact our experiments show that even for moderate sized domains of about a 100 values Korat can take hours to do the computations. It will therefore not be able to scale to real-world examples.

Although a Korat-approach cannot efficiently handle programs with large data domains, with symbolic analysis when symbolic structure is taken as input is almost always modular, i.e. only for one method (that might call others). This means we never really analyze large programs in this fashion in the first place. Secondly, one can often learn a great deal about the execution probabilities from small data domains. We will briefly elaborate on a new approach to handle data within the Korat approach in Section 5.1, that could alleviate Korat’s scalability issues.

The work in this thesis has two main shortcomings we would like to acknowledge: the Korat approach is not completely automated and this in turn meant we only evaluated our solutions on one container class.

5.1 Future Work

When calculating the execution probabilities of code, one should also incorporate a usage profile for the code. Adding usage profiles for integer domains was shown in [39], but how to specify them for structural domains has not been addressed. An example of a usage profile for structures could be that a list will be acyclic 75% of the time. We believe an approach based on separation logic [84] to specify usage profiles for structures is worth investigating. Separation logic is an extension of Hoare logic and was specifically created to reason about programs with references/pointers.

The biggest current issue we face is that Korat is not efficient enough to count structures with large data domains. We believe this problem can be overcome with a combination of Korat to count structures and LattE to count the data constraints. One can think of this as LattE counting the data domains for each structure Korat enumerates. Although this might sound very expensive we believe that with a good caching scheme for the counts, we should get much reuse from previous computations. Some initial experiments shows very promising results.

There are many other directions to extend probabilistic symbolic execution, for example to handle other data domains like floating point values and strings. For floating point values one would need to use a sampling approach, since precise solutions (for example numerical or symbolic integration) will not be feasible in general. Strings can be handled by either counting the size of the language recognized by automata representing string operations, or, by counting solutions to SAT problems (when the string operations are converted to bit vector operations). SPF have back-ends for both automata and bit vectors. Another very interesting direction for extension is to look at concurrent programs and calculating

execution probabilities for them. Here one would need to look at interleavings of program statements from concurrently executing threads and consider the probability for a thread switch. Some initial experiments here shows that for a completely fair scheduler (where every thread switch is equally likely), concurrency errors such as race violations are not nearly as rare as one would think. Of course real-world schedulers are not fair, so we should still consider more realistic scheduling policies.

Appendix A

Appendix

This appendix presents the code for the Binary Search Tree example used throughout the thesis. Specifically we show the code for the **add**, **find** and **delete** methods including the location annotations.

A.1 Code for the Add method

```
1 public void add(int x) {
2     NewNode current = root;
3     if (root == null) {
4         root = new NewNode(x);
5         intendedSize++;
6         return;
7     }
8     while (current.value != x) {
9         if (current.value > x) {
10             if (current.left == null) {
11                 //Location 1
12                 current.left = new NewNode(x);
13                 intendedSize++;
14             } else {
15                 //Location 2
16                 current = current.left;
17             }
18         } else {
19             if (current.right == null) {
20                 //Location 3
21                 current.right = new NewNode(x);
22                 intendedSize++;
23             } else {
24                 //Location 4
25                 current = current.right;
26             }
27         }
28     }
29 }
```

```

27     }
28 }
29 }

```

A.2 Code for the Find method

```

1 public boolean find(int x) {
2     NewNode current = root;
3     while (current != null) {
4         if (current.value == x) {
5             //Location 5
6             return true;
7         }
8         if (x < current.value) {
9             //Location 6
10            current = current.left;
11        } else {
12            //Location 7
13            current = current.right;
14        }
15    }
16    return false;
17 }

```

A.3 Code for the Delete method

```

1 public boolean delete(int x) {
2     NewNode current = root;
3     NewNode parent = root;
4     boolean isLeftChild = true;
5     if (current == null)
6         return false;
7     while(current.value != x) {
8         //assign parent to current
9         parent = current;
10        if (current.value > x) {
11            //Location 8
12            isLeftChild = true;
13            current = current.left;
14        } else {
15            //Location 9
16            isLeftChild = false;
17            current = current.right;
18        }
19        if(current == null) {
20            //Location 10
21            return false;
22        }
23    }
24    if(current.left == null && current.right == null) {
25        if(current == root) {

```

```

26      //Location 11
27      root = null;
28  } else if(isLeftChild) {
29      //Location 12
30      parent.left = null;
31  } else {
32      //Location 13
33      parent.right = null;
34  }
35  } else if(current.right == null) {
36      if(current == root) {
37          //Location 14
38          root = current.left;
39      } else if(isLeftChild) {
40          //Location 15
41          parent.left = current.left;
42      } else {
43          //Location 16
44          parent.right = current.left;
45      }
46  } else if(current.left == null) {
47      if(current == root) {
48          //Location 17
49          root = current.right;
50      } else if(isLeftChild) {
51          //Location 18
52          parent.left = current.right;
53      } else {
54          //Location 19
55          parent.right = current.right;
56      }
57  } else {
58      NewNode successor = getSuccessor(current, x);
59      if(current == root) {
60          root = successor;
61      } else if(isLeftChild) {
62          parent.left= successor;
63      } else {
64          parent.right = successor;
65      }
66      successor.left = current.left;
67  }
68  intendedSize--;
69  return true;
70 }
71
72 private NewNode getSuccessor(NewNode delNode) {
73     NewNode successorParent = delNode;
74     NewNode successor = delNode;
75     NewNode current = delNode.right;
76     while(current != null) {
77         successorParent = successor;
78         successor = current;
79         current = current.left;

```

```
80  }
81  if(successor != delNode.right) {
82      successorParent.left = successor.right;
83      successor.right = delNode.right;
84  }
85  return successor;
86 }
```

Appendix B

Appendix

B.1 Basics of Probability Theory

In this Section, we discuss some notions on probability theory of a finite and countable sample space which will be used in this study. The focus is simply on probability distribution and conditional probability theories.

Definition B.1.1 *Let Ω be the sample space of the experiment [78], that is to say, the set of all possible experiments.*

Let E be a random variable which denotes the value of the outcome for an event. A random variable is referred to as an expression whose value is the outcome of a particular event [78]. Let the random variable represents the roll of one die. For instance, we roll a die of 6 sides and this may produce the possible outcomes 1,2,3,4,5 and 6. We assign the probabilities to the possible outcome of this event, this is called a probability function.

Theorem B.1.2 *The probability, $P(E)$ is assigned to the event E by a distribution function on a sample space which we assumed to be finite and countable Ω , and satisfies the following properties [78];*

1. $P(E) = \frac{\text{Number of outcomes corresponding to event } E}{\text{Total number of outcomes}}$
2. $P(E) \geq 0$
3. $P(\Omega) = 1$, that is to say that the total sum of the probabilities for all values of Ω , is equal to 1

4. If A and B are disjoint subsets of Ω , then

$$P(A \cup B) = P(A) + P(B) \text{ with } A \cap B = \phi, \text{ for all events } A, B \subseteq \Omega$$

Assume we assign a distribution function to a sample space and learn that an event B occurred. How should we change the probability of a remaining event A ?

We refer the new probability for event A as the conditional probability of an event A given an event B and denote it by $P(A|B)$, which is given by:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}; \text{ where } P(B) \neq 0$$

List of References

- [1] *Choco: Java constraint solver (2012)*. <http://choco.emn.fr>.
- [2] *Java Pathfinder Online Available*. <http://babelfish.arc.nasa.gov/trac/jpf/wiki>.
- [3] *Java PathFinder Tool-set*. <http://javapathfinder.sourceforge.net/>.
- [4] *M. UC Davis. Latte integrale*. <http://www.math.ucdavis.edu/~latte>.
- [5] *Symbolic PathFinder – Tool Documentation*. <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc/doc>.
- [6] *Top 10 Software Failures of 2011*. <http://http://www.businesscomputingworld.co.uk/top-10-software-failures-of-2011/>.
- [7] *Toyota says software glitch in data boxes can give faulty speed readings*. <http://www.autoweek.com/article/20100914/CARNEWS/100919945#ixzz1JHXyin8Q>.
- [8] Saswat Anand. Techniques to facilitate symbolic execution of real-worlds programs. PhD dissertation, Georgia Institute of Technology, August 2012.
- [9] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependence analysis and program transformation for symbolic execution. *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems (TACAS)*, pages 117–133, 2007.
- [10] Saswat Anand, Corina Păsăreanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. *In 13th International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.
- [11] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Symbolic execution with abstract subsumption checking. *Proceedings of the 13th International SPIN Workshop on Model checking of Software*, 3925:163–181, 2006.
- [12] Clark Barrett and Cesare Tinelli. CVC3. *In Proceedings of the 19th International Conference on Computer Aided Verification*, 4590:298–302, July 2007.

- [13] A. Barvinok and J. Pommersheim. An algorithmic theory of lattice points in polyhedra. *In New Perspectives in Algebraic Combinatorics*, 38:91–147, 1999.
- [14] Nels E. Beckman and Aditya V. Nori. Probabilistic modular and scalable inference of typestate specifications. pages 211–221, June 2011.
- [15] B Beizer. *Software testing techniques*. International Thomson Computer Press, 1990.
- [16] E. Birnbaum and E. L. Lozinskii. The good old davis-putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.
- [17] Bernard Boigelot and Louis Latour. Counting the solutions of presburger equations without enumerating them. *Theoretical Computer Science*, 313(1):17–29, 2004.
- [18] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. *SIGSOFT. Software Engineering Notes*, 27(4):123–133, 2002.
- [19] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. *In Proceedings of the 31st International Conference on Software Engineering*, pages 144–154, May 2009.
- [20] C Cadar, D Dunbar, and D Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *In Symposium on Operating Systems Design and Implementation*, 8(1):209–224, 2008.
- [21] C. Cadar, P.Godefroid, S. Khurshid, C.S Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. *In Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071, 2011.
- [22] Christian Cadar and Koushik Sen. Symbolic execution for software testing : Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.
- [23] R. Cheung. A user-oriented software reliability model. *IEEE Trans.Soft. Eng.*, 6(2):118–125, 1980.
- [24] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, March 2011.
- [25] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.
- [26] L. A. Clarke. Test data generation and symbolic execution of programs as an aid to program validation. PhD thesis, University of Colorado at Boulder, 1976.

- [27] L.A Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.
- [28] Philippe Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. *Proceedings of the 10th international conference on Supercomputing*, pages 278–285, 1996.
- [29] Marcelo d’Amorim. Efficient explicit-state model checking for programs with dynamically allocated data. PhD Dissertation, University of Illinois at Urbana-Champaign, 2007.
- [30] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.
- [31] de Moula L. and N. Bjørner. Z3: An efficient SMT solver. *In Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 4963:337–340, 2008.
- [32] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9):900–910, 1991.
- [33] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. *In ASE’06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 157–166, 2006.
- [34] Xianghua Deng, Jooyong Lee, and Robby. Efficient symbolic execution algorithms for programs manipulating dynamic heap objects. Technical report SAnToS-TR2009-09-25, Kansas State University, October 2009.
- [35] Xianghua Deng, Jooyong Lee, and Robby. Efficient and formal generalized symbolic execution. *Automated Software Engineering*, 19:233–301, June 2011.
- [36] Xianghua Deng, Robby, and John Hatcliff. Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems. *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, pages 3–12, 2007.
- [37] M. Dowson. The ariane 5 software failure. *ACM SIGSOFT*, 22(2):84, March 1997.
- [38] Matthew B. Dwyer, James C. Corbett, John Hatcliff, Stefan Sokolowski, and Hognjun Zheng. Slicing multi-threaded Java programs: A case study. Technical report KSU CIS TR 99-7. Technical report, Kansas State University, 1999.
- [39] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. Reliability analysis in Symbolic Pathfinder. *In proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 622–631, May 2013.

- [40] Jon William Freeman. Improvements to propositional satisfiability search algorithms. *In Proceedings of AAAI-06: 21st National Conference on Artificial Intelligence*, 1995.
- [41] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. *In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*., pages 166–176, 2012.
- [42] Jaco Geldenhuys, Nazareno Aguirre, F. Marcelo Frias, and Willem Visser. Bounded lazy initialization. *In Proceedings of the 5th International Symposium, NFM 2013*, 7871:229–243, May 2013.
- [43] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. *In Proceedings of AAAI-06: 21st National Conference on Artificial Intelligence*, pages 54–61, July 2006.
- [44] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *handbook of satisfiability. Frontiers in Artificial Intelligence and Applications*, 185:633–654, 2009.
- [45] C. P. Gomes, J. Hoffmann A. Sabharwal, and B. Selman. From sampling to model counting. *In IJCAI*, pages 2293–2299, 2007.
- [46] Orna Grumberg, Assaf Schuster, and Avi Yadgar. *Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis*. Technion, Haifa, Israel, 2004.
- [47] S. Gulwani. Speed: Symbolic complexity bound analysis. *In Proceedings of the 21st International Conference Computer Aided Verification*, pages 51–62, 2009.
- [48] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. *In Proc. of the PLDI’08*, pages 281–292, 2008.
- [49] Tihomir Gvero, Milos Gligoric, Steven Lauterburg, Marcelo d’Amolin, Darko Marinov, and Sarfraz Khurshid. State extensions for Java PathFinder. *ACM*, May 2008.
- [50] Burnim Jacob and Sen Koushik. CREST: Automatic test generation tool for C. Technical report, Kansas State University, <http://code.google.com/p/crest/>.
- [51] K. Jayaraman, D. Harvison, V. Ganeshan, and A. Kiezun. A concolic white-box fuzzer for Java. *In NASA Formal Methods Symposium*, pages 121–125, 2009.
- [52] Khknen K., Launiainen T., Saarikivi O., Kauttio J., Heljanko K., and Niemel I. LCT: An open source concolic testing tool for java programs. *In Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, pages 75–80, 2011.
- [53] Kari Kähkönen. Evaluation of Java PathFinder Symbolic Execution Extension. June 2007.

- [54] Sarfraz Khurshid, C. S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. *In Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TCAS)*, 2619:553–568, 2003.
- [55] Andrew King. Distributed parallel symbolic execution. Master’s thesis, Kansas State University, 2005.
- [56] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [57] Jean-Louis Lassez and ken Mcaloon. Canonical form for generalized linear constraints. *J. Symbolic Computation*, 13:1–24, 1992.
- [58] G. Li, I. Ghosh, and S.P. Rajan. Klover: A symbolic execution and automatic test generation tool for C++ programs. *In Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 609–615, 2011.
- [59] T. Lindholm and F. Yellin. *Java(TM) Virtual Machine Specification*. Prentice Hall, 1999.
- [60] Gary Lindstrom, Peter C. Mehlitz, and Willem Visser. Model checking real time Java using Java PathFinder. *International Journal of Foundations of Computer Science*, pages 444–456, 2005.
- [61] S. Liu and J. Zhang. Program analysis: From qualitative analysis to quantitative analysis. *In Proceedings of the 33rd International Conference on Software Engineering-NIER Track*, pages 956–959, May 2011.
- [62] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symb. Comput*, 38(4):1273–1302, 2004.
- [63] J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, and J. Wu. Software for exact integration of polynomials over polyhedra. *math.MG*, 2011.
- [64] Robin Lohfert, James Lu, and Zhao Dongfang. Solving SQL constraints by incremental translation to SAT. *In Proceedings of the 21st international conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems: New Frontiers in Applied Artificial Intelligence*, pages 669–676, 2008.
- [65] F. Ma, S. Liu, and J. Zhang. Volume computation for boolean combination of linear arithmetic constraints. *In Proceedings of the 22nd International Conference on Automated Deduction*, pages 453–468, August 2009.
- [66] Darko Marinov. Automatic testing of software with structurally complex inputs. PhD thesis, Massachusetts Institute of Technology, February 2005.

- [67] Jean Philippe Martin. Upper and lower bounds on the number of solutions. Technical report MSR-TR-2007-164, Microsoft Research Cambridge, December 2007.
- [68] Saša Misailović, Aleksandar Milićević, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. Parallel test generation and execution with Korat. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 135–144, 2007.
- [69] D. Monniaux. Abstract interpretation of probabilistic semantics. *In Proceedings of the 7th International Static Analysis Symposium*, pages 322–339, June 2000.
- [70] A. Morgado, P. Matos, V. Manquinho, and J. Marques-Silva. Counting models in integer domains. Technical report 05/2006, INESC-ID, March 2006.
- [71] Carroll Morgan and Annabelle McIver. pGCL: Formal reasoning for random algorithms. *South African Comp Jnl*, 22:14–27, 1999.
- [72] G.J. Myers. Art of software testing. *John Wiley and Sons Inc*, 1979.
- [73] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. *Planning Report 02-3*, (7007.011), May 2002.
- [74] Pavel Parízek and Ondřej Lhoták. Identifying future field accesses in exhaustive state space traversal. *26th International Conference on Automated Software Engineering (ASE)*, pages 93–102, 2011.
- [75] Erin Parker and Siddhartha Chatterjee. An automata-theoretic algorithm for counting solutions to presburger formulas. pages 104–119, 2004.
- [76] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. *In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, 2008.
- [77] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. *In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 504–515, 2011.
- [78] Wiebe R. Pestman. *Mathematical Statistics*. ser. De Gruyter Texbook, 2009.
- [79] C. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. *In Proceedings of SPIN’04*, 2989, 2004.
- [80] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: Symbolic execution of Java byte-code. *In Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE*, pages 179–180, 2010.

- [81] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, pages 1–35, 2013.
- [82] William Pugh. Counting solutions to presburger formulas: How and why. Technical report, October 1998.
- [83] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, September 1994.
- [84] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
- [85] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.
- [86] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8:50–64, 2010.
- [87] T. Sang, P. Beame, and H. A. Kautz. Performing bayesian inference by weighted model counting. In *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference*, pages 475–482, July 2005.
- [88] Junaid Haroon Siddiqui. Improving systematic constraint-driven analysis using incremental and parallel techniques. PhD dissertation, University of Texas at Austin, May 2012.
- [89] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. *ISSTA*, pages 183–193, 2010.
- [90] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. in *International Conference on Tests and Proofs*, 4966:134–153, April 2008.
- [91] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *L. L. Pollock and M. Pezzè, editors, ISSTA*, pages 37–48, 2006.
- [92] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Jnl*, 10(2):203–232, 2003.
- [93] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pages 97–107, July 2004.
- [94] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for red-black trees using abstraction. In *David F. Redmiles, Thomas Ellman and Andrea Zisman, editors, ASE*, pages 414–417, 2005.

- [95] Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. *American Association for Artificial Intelligence*, 2004.
- [96] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. *In Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, 2005.
- [97] Hong Yu, Huang Song, Liu Xiaoming, and Yu Xiushan. Using symbolic execution in embedded software testing. *International Conference on Computer Science and Software Engineering*, 2008.
- [98] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.